Neue Ansätze zur Speicherzugriffsanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Olaf Krzikalla
geboren am 22. Mai 1970 in Löbau

Gutachter:
Prof. Dr. rer. nat. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr. techn. Dieter Kranzlmüller, Ludwig-Maximilians-Universität München

Tag der Einreichung: 14. Juni 2018
Tag der Verteidigung: 25. Oktober 2018

Dresden, den 29. Oktober 2018

Kurzfassung

Vor dem Hintergrund stetig wachsender Anforderungen an skalierbare Architekturen und Programme im Hochleistungsrechnen haben sich in den letzten Jahren neue Programmiermodelle als Alternativen zu den bisher verwendeten Nachrichten-basierten Modellen etabliert. Verteilte Anwendungen können heutzutage durch asynchrone, einseitige Speicherzugriffe auf nicht knotenlokale Speicherbereiche zugreifen, so dass sich ein gemeinsam genutzter Adressraum über alle Knoten aufspannt. Das darauf aufsetzende PGAS-Programmiermodell eröffnet neue Wege zur Entwicklung leistungsfähiger Programme, birgt aber auch neue Herausforderungen zur Sicherstellung der Programmkorrektheit und -effizienz.

Diese Dissertation leistet einen Beitrag zum systematischen Verständnis von parallelen verteilten Anwendungen mit gemeinsam genutztem Adressraum. Der Fokus liegt dabei auf der Analyse des Zusammenspiels von asynchronen und synchronen Speicherzugriffen in diesem Adressraum. Das in der Dissertation vorgestellte Konzept des Speicherzugriffsdiagramms erschließt dem Programmierer eine neue Analyseperspektive. Das zugrunde liegende Task-Graph-Modell wurde erweitert, so dass die kausalen Beziehungen zwischen asynchronen Speicherzugriffen und anderen Programmereignissen präzise abgebildet werden. Durch eine Anpassung des Modells kann für die im allgemeinen Fall NP-vollständige Berechnung von Synchronisationsbeziehungen in einem Task Graphen ein Algorithmus angegeben werden, der diese Berechnung in quasi-linearer Zeit durchführt.

Die Dissertation demonstriert die Anwendung der entwickelten Methoden an mehreren Beispielen aus der Forschungspraxis. Dabei wird die Zweckmäßigkeit von Speicherzugriffsdiagrammen zum Veranschaulichen der logischen Ursachen von Programmfehlern oder Performance-Schwachstellen deutlich. Ein weiteres Resultat der Dissertation ist der Prototyp eines neuartigen Analysewerkzeugs, welches bereits in mehreren Forschungseinrichtungen genutzt wird.

Abstract

Due to the ever-increasing demand for scalable architectures and programs in the field of high performance computing new programming models have been established in the last years as alternatives to the message-based programming models. Today, distributed applications can create a shared global address space over all nodes of an HPC system by using asynchronous, one-sided memory accesses to remote memory. Based on this global address space the PGAS programming model has emerged, which enables new ways to develop powerful applications, but also introduces new challenges to ensure program correctness and efficiency.

This thesis contributes to the systematic understanding of parallel distributed applications with a shared global address space. It focuses on the analysis of the interaction of asynchronous and synchronous memory accesses in this address space. The concept of memory access diagrams presented in this thesis opens up a new analysis perspective to the programmer. The underlying task graph model has been enhanced to accurately map the causal relationships between asynchronous memory accesses and other program events. By adapting the model, an algorithm can be specified for the generally NP-complete calculation of synchronization relationships, which performs this calculation in quasi-linear time.

The thesis demonstrates and evaluates the application of the new methods with various examples from research practice. The demonstration illustrates the usefulness of memory access diagrams to visualize the logical causes of programming errors and performance flaws. A further result of the thesis is the prototype of a novel analysis tool, which is already in use in several research institutions.

Inhaltsverzeichnis

1 Einführung

> *Programming model will be necessary: heroic compilers will not be able to hide the level of concurrency from applications. [Exa16]*

Softwareentwicklung ist inhärent komplex. Dieses Problem ist spätestens seit der Prägung des Begriffs „Softwarekrise“ omnipräsent in der Informatik. Eine Bewältigung ist nur durch einen iterativen Prozess, bestehend aus Programmsynthese und Programmanalyse möglich. Programmiersprachen, Bibliotheken und Programmiermodelle erlauben dem Softwareentwickler, sich bei der Programmsynthese auf die Formulierung des Problems zu konzentrieren. Die Programmanalyse beginnt bereits bei der Übersetzung mit Hilfe typ-überprüfender Compiler. Außerdem finden explizite Analysewerkzeuge wie z. B. Debugger oder Profiler Anwendung, um die Korrektheit und Effizienz eines Programms untersuchen zu können.
Auf dem Gebiet des Hochleistungsrechnens werden in Zukunft neue Herausforderungen zu bewältigen sein, um den stetig wachsenden Anforderungen an skalierbare Architekturen und Programme gerecht zu werden. Insbesondere wird die Entwicklung neuer Technologien notwendig sein, um die Exascale-Schranke zu durchbrechen [Exa16]. Im Bereich der Programmiermodelle hat diese Entwicklung bereits begonnen. Die bisher verwendeten Nachrichten-basierten Modelle werden abgelöst von Ansätzen, mit denen direkt auf Speicherbereiche in entfernten Knoten zugegriffen werden kann. Ein solcher Ansatz ist das PGAS-Programmiermodell, welches das Entwickeln von verteilten Anwendungen mit einem gemeinsam genutzten Adressraum erlaubt. Entsprechende Programmierschnittstellen wie z. B. GASPI [GAS11], OpenSHMEM [OPE13] oder one-sided MPI [For12] werden sehr dynamisch weiterentwickelt. Diese Schnittstellen setzen das Konzept eines gemeinsam genutzten Adressraumes um, indem sie asynchrone, einseitige Zugriffsfunktionen auf nicht knoten-lokale Speicherbereiche bieten. Die damit entstehenden neuen Möglichkeiten der Programmsynthese erfordern wiederum auch eine Weiterentwicklung der Programmanalyse.
In der vorliegenden Arbeit werden Analysemethoden entwickelt, welche neue Perspektiven auf parallele Anwendungen mit gemeinsam genutztem Adressraum erschließen. Der Fokus liegt dabei auf der Betrachtung der Speicherzugriffe auf den gemeinsam genutzten Adressraum einer parallelen Anwendung. Durch die potentielle Interaktion synchroner und asynchroner Speicherzugriffe in solchen Anwendungen steigt auch die Komplexität der Softwareentwicklung. Damit werden auch Werkzeuge notwendig, die eine Analyse der entwickelten Anwendungen hinsichtlich Korrektheit und Effizienz ermöglichen.
Der wissenschaftlich-technische Beitrag der Arbeit kann in zwei Teile gegliedert werden. Im ersten Teil wird ein Modell entwickelt, welches die Ereignisse in einem Programm mit gemeinsam genutztem Adressraum in ihren logischen Zusammenhängen präzise darstellen kann. Im zweiten Teil werden auf der Grundlage des entwickelten Modells mehrere Anwendungsmöglichkeiten und darauf aufbauende Werkzeuge zur Analyse entsprechender Programme konzipiert. Diese

Werkzeuge umfassen eine Korrektheitsanalyse zum zuverlässigen Finden nichtdeterministischer Konflikte, eine Effizienzanalyse und eine Methode zur gemeinsamen Visualisierung von synchronen und asynchronen Speicherzugriffen. Die Evaluierung der Methoden und Werkzeuge erfolgt anhand von GASPI-Anwendungen. Die verwendeten Prinzipien sind jedoch auch auf ähnliche Programmierschnittstellen übertragbar.
Die Dissertation ist wie folgt aufgebaut: im weiteren Verlauf dieses Kapitels wird eine Einführung in das PGAS-Programmiermodell gegeben und die Bedeutung von Speicherzugriffen in parallelen Programmen erläutert. In Kapitel 2 werden typische Herausforderungen bei der Analyse paralleler Programme erläutert. Weiterhin enthält Kapitel 2 eine kurze Beschreibung der GASPI-Programmierschnittstelle. Kapitel 3 präsentiert eine Übersicht über bereits existierende Methoden und Werkzeuge, die mit den in dieser Arbeit behandelten Themen in Zusammenhang stehen. In Kapitel 4 wird ein Task-Graph-Modell eingeführt, mit dem eine hybrid parallele Programmausführung abgebildet werden kann. Dieses Modell abstrahiert die Programmereignisse einer Anwendung mit gemeinsam genutztem Adressraum und repräsentiert die sich ergebenden kausalen Ordnungen von synchronen und asynchronen Speicherzugriffen auf diesen Adressraum. Kapitel 5 stellt aufbauend auf dem Task-Graph-Modell das Konzept des Speicherzugriffsdiagramms zur Analyse von PGAS-Programmen vor. Speicherzugriffsdiagramme veranschaulichen die Interaktionen von asynchronen und synchronen Speicherzugriffen. In Kapitel 6 wird mit einem prototypisch realisierten Werkzeug die Anwendung der entwickelten Konzepte anhand der Analyse von GASPI-Anwendungen aus der Forschungspraxis demonstriert. Das abschließende Kapitel 7 fasst die Ergebnisse der Arbeit zusammen und gibt einen Überblick über mögliche Folgearbeiten.

1.1 Das PGAS-Programmiermodell

Das PGAS (Partitioned Global Address Space [PGA13]) Programmiermodell basiert auf einem abstrakten gemeinsamen Adressraum in einem verteilten System. In diesem Programmiermodell kann ein Prozess auch auf Speicherbereiche entfernter Knoten direkt lesend und schreibend zugreifen. Für alle anderen Prozesse, insbesondere für den Prozess auf dem Zielknoten, sind solche Zugriffe transparent. Moderne Hardware erlaubt es zudem, diese Zugriffe asynchron zum aktiven Prozess auszuführen. In dem daraus resultierenden einseitigen asynchronen Kommunikationsmodell sind zum einen Datentransfer und Prozess-Synchronisation entkoppelt, zum anderen können Berechnung und Kommunikation gleichzeitig erfolgen.
Abbildung 1.1 stellt das Speichermodell eines PGAS-Systems dar. Jeder PGAS-Prozess besitzt Speichersegmente, auf die nur lokal zugegriffen werden kann. Hinzu kommen Speichersegmente, welche einem gemeinsamen Speicher zugeordnet sind. Ein gemeinsames Speichersegment ist partitioniert, wobei jedem Prozess statisch die Partition zugeordnet ist, die den knoten-lokalen Speicher repräsentiert. Entsprechend werden die einem Prozess zugeordneten Partitionen lokale Partitionen genannt, alle anderen Partitionen sind Remote-Partitionen. Lokale Variablen sind entweder im lokalen Speichersegment oder in der lokalen Partition eines gemeinsamen Speichersegmentes abgelegt. Remote-Variablen sind immer in Remote-Partitionen eines gemeinsamen Speichersegmentes abgelegt.

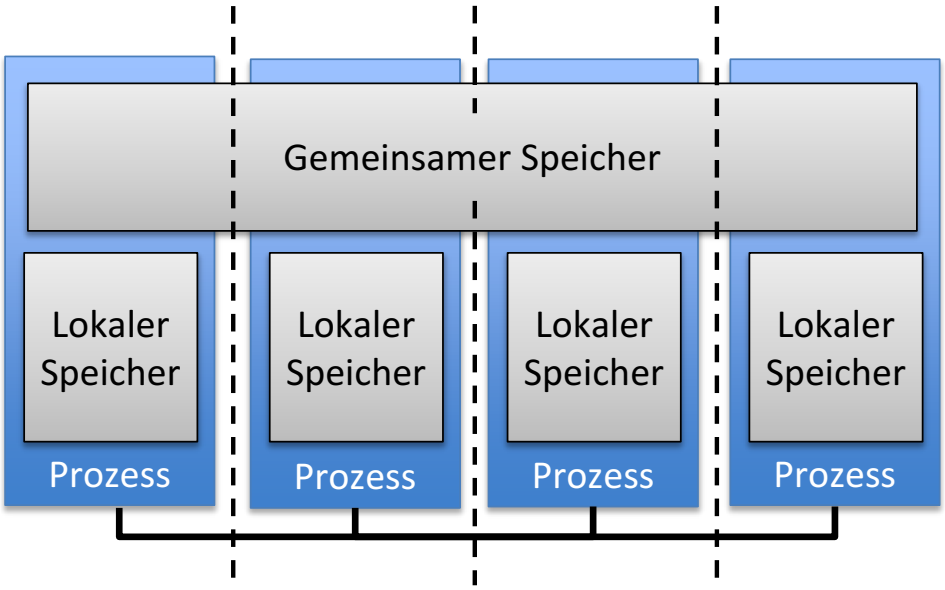


Abbildung 1.1: Im PGAS-Speichermodell wird der gemeinsame Speicherbereich entlang der Prozess- bzw. Knotengrenzen partitioniert

Der Zugriff auf Remote-Variablen erfolgt durch einseitige Kommunikationsoperationen. Die zwei fundamentalen Primitive sind die *get*- und die *put*-Operation. Die get-Operation kopiert Variablen von einer Remote-Partition in die lokale Partition. Im Gegensatz dazu kopiert die put-Operation Variablen von einer lokalen Partition in eine Remote-Partition. Für beide Operationen gilt, dass nur der die Operation initiierende Prozess – der lokale Prozess – *aktiv* ist. Für den der Remote-Partition zugeordneten Prozess ist der Datentransfer transparent. Dieser Remote-Prozess muss sich zu keiner Zeit aktiv an der Kommunikationsoperation beteiligen und wird auch nicht implizit über einen stattfindenden Datentransfer benachrichtigt.

Die Synchronisation der Zugriffe auf Remote-Variablen erfolgt getrennt von den Kommunikationsoperationen. Der aktive Prozess kann überprüfen, ob eine oder mehrere zuvor gestartete Kommunikationsoperationen lokal abgeschlossen sind. Ist eine get-Operation lokal abgeschlossen, so stehen alle angeforderten Variablen in der lokalen Partition bereit. Aufgrund der Datenabhängigkeit sind damit auch die Lesezugriffe auf Remote-Variablen abgeschlossen. Ist eine put-Operation lokal abgeschlossen, so bedeutet das jedoch nur, dass die Lesezugriffe auf die zu kopierenden Variablen in der lokalen Partition abgeschlossen sind. Es bedeutet nicht, dass diese Variablen auch schon auf der Remote-Partition geschrieben wurden. Diese Überprüfung kann vom Remote-Prozess mittels einer *test*-Operation durchgeführt werden. Auch die test-Operation ist eine einseitige Funktion, da eine Beteiligung des ursprünglich die put-Operation auslösenden Prozesses nicht notwendig ist.

Aufgrund des Zugriffs auf Remote-Variablen durch explizite Transferoperationen ist ein das Gesamtsystem umfassendes Cache-Kohärenzprotokoll nicht notwendig. PGAS-Systeme sind dadurch höher skalierbar als klassische Systeme mit gemeinsam genutztem Speicher [DAS12, Kapitel 5.5],[Pat11, Kapitel 5.4]. Durch die Entkopplung von Datentransfer und Synchronisation kann aber auch gegenüber Nachrichten-basierten Systemen eine höhere Skalierbarkeit erreicht werden [SSO+95, BCA+06, GPI12]. Asynchrone Datentransfers erlauben zudem eine Überlap-

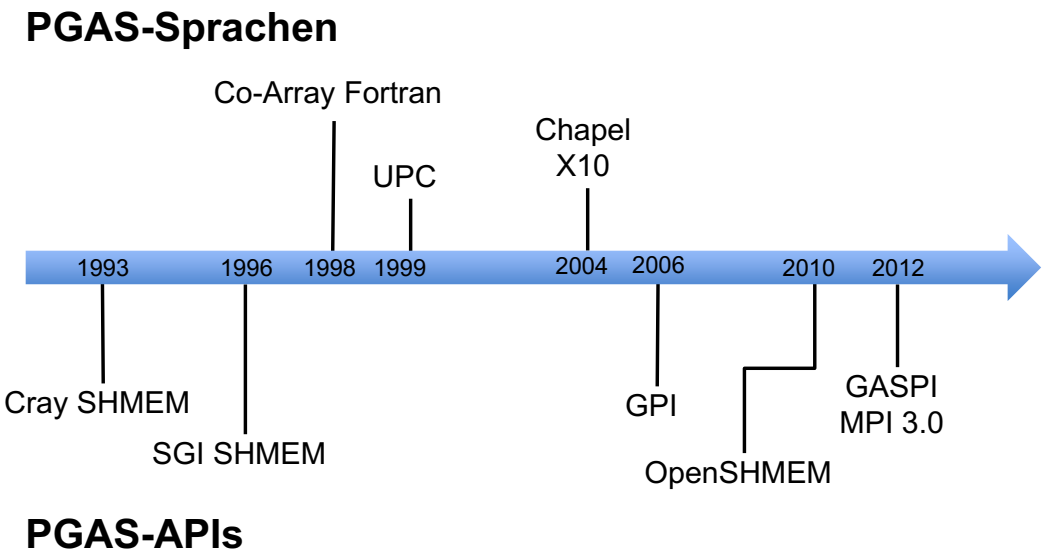


Abbildung 1.2: Die Entwicklungsgeschichte des PGAS-Modells

pung von Kommunikation und Berechnung, wodurch der Einfluss von Netzwerk-Latenzen auf die Programmeffizienz verringert wird. Da trotz des Zugriffs durch explizite Transferoperationen jeder Prozess prinzipiell wahlfreien Zugriff auf alle Variablen im gemeinsamen Speicher besitzt, sind viele Lösungen, aber auch Herausforderungen, die bei der Programmierung von Systemen mit gemeinsam genutztem Speicher auftreten, auch auf PGAS-Systeme übertragbar.

Das PGAS-Modell steht dem Programmierer heutzutage in zwei prinzipiellen Ausprägungen zur Verfügung. Zum einen existiert eine Reihe von PGAS-Sprachen, die meist als Varianten klassischer Programmiersprachen entwickelt wurden. Unified Parallel C (UPC) [CDC+99] ist eine C-Erweiterung. Co-Array Fortran (CAF) [NR98] basiert auf Fortran und wurde inzwischen in Fortran 2008 integriert. Weitere Beispiele sind X10 [CGS+05] und Chapel [CCZ04]. Allen Erweiterungen ist gemein, dass sie die Basissprache um Operationen über verteilte Felder erweitern. Diese verteilten Felder werden im gemeinsamen Speichersegment abgelegt und der Programmierer kann auf dem syntaktisch üblichen Weg auf die Feldelemente zugreifen. Bei Zugriffen auf Feldelemente in Remote-Partitionen werden vom Compiler die nötigen Kommunikationsoperationen automatisch eingefügt.

Parallel zu den PGAS-Sprachen haben sich auch PGAS-APIs entwickelt, die die beschriebenen einseitigen Kommunikationsoperationen bereitstellen. Anders als in PGAS-Sprachen muss der Programmierer bei der Nutzung solcher APIs die Kommunikationsoperationen explizit aufrufen. Das erhöht zwar den Programmieraufwand, allerdings kann in der Praxis häufig eine höhere Programmeffizienz gegenüber den PGAS-Sprachen erreicht werden, da Kommunikationszeitpunkte präzise bestimmt werden können und auch keine versteckte Kommunikation stattfindet. Beispiele für PGAS-APIs sind GASPI [GAS11], SHMEM [BK94] und der aus SHMEM entwickelte, offene Standard OpenSHMEM [CCP+10, OPE13]. Ein weiteres Beispiel sind die in der MPI-3-Spezifikation hinzugefügten Routinen für einseitige Kommunikation [For12], womit auch MPI das PGAS-Programmiermodell unterstützt. Abbildung 1.2 gibt einen zeitlichen Überblick über die Entwicklung von PGAS-Sprachen und PGAS-APIs. Es ist deutlich zu erkennen, dass die Entwicklung von PGAS-APIs in den letzten Jahren an Fahrt aufgenommen hat.

1.2 Die Bedeutung von Speicherzugriffen in der Programmierung

Der Speicher ist ein fundamentales Konzept der Informatik. Während einer Programmausführung durchläuft ein Computer eine Vielzahl von Zuständen, die im Speicher abgelegt werden. Eine besondere Bedeutung kommt dabei dem Arbeits- bzw. Hauptspeicher zu, auf dessen Speicherzellen wahlfrei zugegriffen werden kann und in dem volatile, d.h. sich schnell ändernde Daten abgelegt werden. Somit finden Zugriffe in den Hauptspeicher sehr häufig statt. Aufgrund dessen stellen diese Zugriffe einen herausragenden Faktor für die Effizienz eines Programms dar.
Der Begriff *memory wall* [WM95] bezeichnet die immer größer werdende Lücke zwischen der Verarbeitungsgeschwindigkeit eines Prozessors und der Zugriffsgeschwindigkeit auf den Hauptspeicher. Um Zugriffe auf den Hauptspeicher effizient zu gestalten, wurde das Konzept der Speicherhierarchien entwickelt [Pat11, Kap. 2]. Die heutzutage wichtigste und im HPC-Bereich durchgängig verwendete Technik ist der Einsatz von CPU-Caches. Trotzdem stellt die *memory wall* insbesondere in der HPC-Programmierung weiterhin eine der wesentlichsten Herausforderungen dar [RZR+15].
Ein Speicherzugriff im Sinne dieser Arbeit wird bestimmt durch die Speicheradresse, die Größe des Bereichs und den Typ des Zugriffs: lesend oder schreibend. Der gelesene bzw. geschriebene Wert selbst ist für die Charakterisierung eines Speicherzugriffs unerheblich. Abfolgen von Speicherzugriffen werden als Speicherzugriffsmuster bezeichnet. Speicherzugriffsmuster unterscheiden sich hinsichtlich ihrer räumlichen und zeitlichen Lokalitätseigenschaften [Den05]. Sie beeinflussen wesentlich die Effizienz eines CPU-Caches. Die Analyse und Optimierung von Speicherzugriffsmustern stellt inzwischen einen eigenen Forschungszweig dar, in dem sowohl problembezogene Lösungen [Wol89] als auch allgemein verwendbare Analysemethoden [BSGT04, NS07, RB11] entwickelt werden.
In PGAS-Programmen dienen Speicherzugriffe auch zur Kommunikation zwischen den Prozessen. Um eine Überlappung von Berechnung und Kommunikation im PGAS-Modell zu erreichen, müssen demnach auch innerhalb einer Partition Speicherzugriffe zur Berechnung und Speicherzugriffe zur Kommunikation überlappend, d.h. parallel zueinander ausgeführt werden. Diese Speicherzugriffe müssen vom Programmierer so koordiniert werden, dass nicht gleichzeitig auf eine bestimmte Speicherzelle so zugegriffen wird, dass der Programmzustand ungewollt nichtdeterministisch wird. Diese in der Programmierung paralleler Systeme zentrale Herausforderung wird in der PGAS-Programmierung noch größer, da Kommunikation und Synchronisation entkoppelt sind und außerdem Speicherzugriffe auch vollständig asynchron stattfinden können. Darüber hinaus ist in PGAS-Systemen Datenlokalität nicht nur ein wichtiges Laufzeit-, sondern auch ein Energie-Effizienzkriterium [JSC14]. Aus diesen Überlegungen ergeben sich folgende Fragen für die Untersuchung von Speicherzugriffen in PGAS-Programmen:

1. Kann eine Methode zuverlässig feststellen, ob die Koordination der parallelen Speicherzugriffe insofern korrekt ist, dass das Programm nicht ungewollt nichtdeterministisch ist?

2. Welche Erkenntnisse hinsichtlich der Effizienz eines PGAS-Programms können aus der Analyse der Speicherzugriffe gewonnen werden?

3. Können die komplexen Speicherzugriffsmuster dem PGAS-Programmierer so präsentiert

werden, dass dieser daraus weiterführende Einsichten wie z. B. ein besseres Verständnis des Datenflusses erhält?

Frage (1) kann nur mit „ja“ beantwortet werden, wenn die während einer Programmausführung stattfindenden asynchronen Speicherzugriffe in einen logischen Zusammenhang zu den synchronen Speicherzugriffen gebracht werden können. Nur so wird die Zuverlässigkeit der Analyse gewährleistet. Zur Beantwortung von Frage (2) ist eine Betrachtung der Möglichkeiten notwendig, die das PGAS-Programmiermodell für die Entwicklung hoch skalierbarer Anwendungen bietet. Wichtig ist vor allem das Erreichen einer möglichst großen (aber immer noch korrekten) Überlappung von Kommunikation und Berechnung. Die Antwort auf Frage (3) muss eine abstrahierende Darstellung von Speicherzugriffen beinhalten, um eine übersichtliche Präsentation zu erhalten.
In dieser Arbeit werden für alle drei Fragestellungen Methoden entwickelt. Die darauf basierenden Werkzeuge führen eine dynamische Programmanalyse durch, können also eine bestimmte Programmausführung analysieren. Statische Analysewerkzeuge, die diese Informationen direkt aus dem Programmtext extrahieren können, sind nicht Gegenstand der Arbeit. Jedoch kann das in Kapitel 4 entwickelte Modell prinzipiell auch für eine statische Programmanalyse verwendet werden.

2 Problemfeld: Analyse von parallelen Programmen

Das ist ein weites Feld. [Fon12]

Eine Programmanalyse ist die systematische Untersuchung eines Computerprogramms, mit der Aussagen über Eigenschaften des Programms gewonnen werden können. Im Fokus der Analyse stehen dabei meistens die Korrektheit und Effizienz eines Programms. Dieses Kapitel stellt Programmeigenschaften vor, die für die Korrektheit und Effizienz insbesondere von HPC-Programmen von Bedeutung sind. Für viele der dabei verwendeten Begriffe lassen sich unterschiedliche Definitionen angeben, die sich jeweils in Aspekten unterscheiden. In diesen Fällen wurde die für diese Arbeit geeignetste Definition ausgewählt. Außerdem wird ein Überblick über alternative Definitionen in der Literatur gegeben.

2.1 Parallele Programme

Der Begriff der Parallelität wird in der Informatik unterschiedlich definiert. Eine in der englischen Literatur oft verwendete Definition [OA10, Gre06] besagt, dass parallele Verarbeitung der gleichzeitige Einsatz mehrerer Rechenressourcen zur Lösung eines Problems ist. In [Cla06] findet sich dagegen folgende Definition:

Definition 1. *Arbeitsabläufe bzw. deren Einzelschritte heißen* parallel*, wenn sie gleichzeitig und voneinander unabhängig durchgeführt werden können.*

Diese enger gefasste Definition schließt datenparallele Verarbeitung mithilfe von SIMD aus, da im SIMD-Modell die parallelen Arbeitsabläufe zwar gleichzeitig, aber immer im Gleichschritt, also nicht unabhängig voneinander stattfinden. In der vorliegenden Arbeit wird Parallelität im Sinne von Definition 1 verwendet. Ein Programm ist also dann parallel, wenn es mindestens zwei Arbeitsabläufe enthält, die gleichzeitig und voneinander unabhängig durchgeführt werden können. Besteht ein Arbeitsablauf aus einer Sequenz von auf einer CPU ausgeführten Maschinenbefehlen, so wird dieser Ablauf in der Folge als *Thread* bezeichnet. Ein Arbeitsablauf ist jedoch nicht äquivalent zu einem Thread. HPC-Maschinen stellen oft zusätzliche Rechenressourcen, z. B. in Form von RDMA-Controllern, zur Verfügung, die einen Datentransfer unabhängig von CPUs unterstützen. Die von RDMA-Controllern ausgeführten Operationen sind ebenfalls Arbeitsabläufe und können parallel zu den übrigen Arbeitsabläufen stattfinden.
Moderne HPC-Cluster haben eine hierarchische Hardware-Architektur. Einzelne Knoten eines solchen Clusters bestehen aus Multicore- bzw. Manycore-Prozessoren, die zusätzlich durch Beschleuniger-Karten unterstützt werden können. Parallele Programme müssen die Hierarchieebenen dieser Hardware abbilden, um eine effiziente Auslastung der Rechenressourcen zu erreichen [RHJ09]. Aus diesem Grund besitzen heutige HPC-Programme mehrere Parallelitätsebenen. Hybrid parallele Programme bestehen aus einer Menge von Prozessen, die jeweils aus

mehreren Threads bestehen. Läuft ein Teil der Threads auf Beschleuniger-Karten, so ist das Programm heterogen parallelisiert. Im Gegensatz dazu bestehen alle Prozesse homogen parallelisierter Programme aus jeweils genau einem Thread.
Von der Parallelität abzugrenzen ist der Begriff der Nebenläufigkeit (nach [Cla06]):

Definition 2. *Arbeitsabläufe bzw. deren Einzelschritte heißen* nebenläufig*, wenn sie voneinander unabhängig durchgeführt werden können.*

Parallele Arbeitsabläufe sind demnach auch nebenläufig. Ebenfalls nebenläufig ist aber auch präemptives Multitasking, bei dem unabhängige (Teil-)Probleme auf nur einem Prozessor und damit nicht gleichzeitig bearbeitet werden.

2.2 Parallelität und Determinismus

Der Begriff Determinismus ist für einen Algorithmus bzw. ein Programm nach [Cla06] wie folgt definiert:

Definition 3. *Das Programm heißt* deterministisch*, wenn es zu jeder Programmsituation höchstens eine nachfolgende Situation geben kann, wenn also zu jedem Zeitpunkt der Fortschritt eindeutig bestimmt ist.*

In der Originalquelle wird statt *Programm* der Begriff *Algorithmus* verwendet. Da jedoch in dieser Arbeit Programme als einzige Repräsentanten von Algorithmen behandelt werden, wird vereinfachend der Begriff *Programm* benutzt.
Nach Definition 3 sind alle parallelen Programme nicht-deterministisch, da Situationen existieren, in denen mehrere verschiedene Einzelschritte als nächstes ausgeführt werden können. Aufgrund dessen wird Nichtdeterminismus in Zusammenhang mit parallelen Programmen für verschiedene Konzepte verwendet und parallele Programme werden mit verschiedenen Begriffen in der Literatur (nondeterminism [Kra00], nondeterminacy [EGP92], indeterminacy [LMC87]) charakterisiert. In probabilistischen oder randomisierten Programmen bestimmen Zufallszahlen den Rechenfortschritt. Diese Programme sind ebenfalls nicht deterministisch, werden aber in dieser Arbeit nicht weiter behandelt.
In [RDC00] werden parallele Programme zwischen extern und intern nicht-deterministisch unterschieden. Extern nicht-deterministische Programme liefern bei wiederholter Ausführung für denselben Eingabewert unterschiedliche Resultate. Im Gegensatz dazu liefern intern nicht-deterministische Programme für denselben Eingabewert immer dasselbe Resultat, wobei allerdings der interne Ausführungspfad bei jedem Programmlauf variieren kann. Die Unterscheidung zwischen extern und intern nicht-deterministischen Programmen ist eng verknüpft mit dem Begriff der Determiniertheit (nach [Cla06]):

Definition 4. *Ein Programm ist* determiniert*, wenn zu jedem möglichen Eingabewert höchstens ein Ausgabewert geliefert wird.*

Ist ein möglicher Eingabewert valide, dann wird ein determiniertes Programm immer terminieren und genau einen Ausgabewert liefern. Deterministische Programme sind immer auch

determiniert. Determinierte Programme jedoch dürfen intern nicht-deterministisch sein. Nicht-determinierte Programme dagegen sind auch extern nicht-deterministisch.
Ein korrektes Programm zeichnet sich im Allgemeinen durch Determiniertheit aus. Ist ein paralleles Programm nicht determiniert, dann deutet das oft auf Programmfehler hin. Es existieren jedoch auch gewollte und akzeptierte nicht-determinierte Programme. Beispielsweise kann das Damenproblem [Cla06] so parallelisiert werden, dass das Programm in verschiedenen Läufen verschiedene, jeweils korrekte Lösungen erzeugt. Numerische Auslöschung bei der Akkumulation von parallel erzeugten Teillösungen ist ein weiteres Beispiel für akzeptierte Nicht-Determiniertheit. Auf eine Reduzierung oder Vermeidung der dadurch entstehenden Ungenauigkeiten z. B. mit Hilfe der Kahan-Summierung [Kah65] wird in der Praxis oft verzichtet.

2.3 Wettlaufsituationen

Aus den Definitionen 1 und 3 ergibt sich, dass die Ausführung paralleler Programme *a priori* nicht-deterministisch ist. Eine sinnvolle Charakterisierung eines parallelen Programms muss daher zwischen einer nicht-deterministischen Programmausführung an sich und sich daraus eventuell ergebenden nicht-deterministischen Programmresultaten unterscheiden.
Arbeitet eine Menge deterministischer Arbeitsabläufe auf voneinander unabhängigen Teilen eines Programmzustandes, so wird nach Ausführung aller Arbeitsabläufe ein bestimmter Ausgangszustand immer zu genau einem Endzustand führen, der sich aus der Summe der Zustandsänderungen ergibt. Der von der Menge dieser Arbeitsabläufe bestimmte Programmteil ist also determiniert. Formal kann dieser Sachverhalt z. B. mit der Regel der disjunkten Nebenläufigkeit (disjoint concurrency [O'H07]) beschrieben werden. Davon abzugrenzen sind parallele Arbeitsabläufe, die auf gemeinsamen Teilen des Programmzustandes arbeiten. Ist dabei der Endzustand nach Ausführung dieser Arbeitsabläufe abhängig von der Ausführungsreihenfolge der Arbeitsabläufe, so stehen diese Arbeitsabläufe zueinander in einer *Wettlaufsituation*.

Definition 5. *Eine* Wettlaufsituation *oder* race condition *zwischen zwei oder mehreren nebenläufigen Arbeitsabläufen liegt vor, wenn der resultierende Programmzustand von der Reihenfolge der Ausführung dieser Arbeitsabläufe abhängt.*

Die Definition verwendet den weiter gefassten Begriff der Nebenläufigkeit, da in der parallelen Programmierung durchaus Situationen entstehen, in denen Arbeitsabläufe zwar unabhängig voneinander, jedoch nicht zeitgleich ausgeführt werden können. Solche Arbeitsabläufe können zum Beispiel in kritischen Abschnitten gekapselt sein.
Wettlaufsituationen sind ein essentieller Bestandteil der parallelen Programmierung. Zur Sicherstellung der Korrektheit von Kommunikationen zwischen Threads oder Prozessen müssen diese sich synchronisieren. Die Synchronisation wiederum geschieht durch Ausnutzung von Wettlaufsituationen. Neben der Sicherstellung der Korrektheit werden Wettlaufsituationen auch zum Erreichen von Laufzeit-Effizienz eingesetzt. In einem parallelen Programm mit dynamischer Lastverteilung zum Beispiel sind Wettlaufsituationen zwischen Arbeitspaketen bewusst vorgesehen. Während einer konkreten Abarbeitung reagiert das Programm dann auf die tatsächliche Ausführungsgeschwindigkeit und verteilt die Arbeitspakete entsprechend der Verfügbarkeit der Rechenressourcen.

Im Folgenden werden verschiedene Arten von nicht-deterministischen Konstrukten in parallelen Programmen vorgestellt. Für jede Art wird erläutert, welche Wettlaufsituationen sich ergeben können.

Data Races

Der Zustand eines Programms ergibt sich aus allen zu einem bestimmten Zeitpunkt gespeicherten Daten. Da der Datenspeicher dabei eine zentrale Rolle spielt, wurde für potentielle Wettlaufsituationen diesen Speicher betreffend der Begriff *data race* geprägt. Dieser Begriff wird je nach Anwendungsbereich verschieden definiert [AHB03, ISO12]. Die folgende Definition nach [DP12] verwendet die bereits in dieser Arbeit eingeführten Begriffe.

Definition 6. *Ein* data race *liegt vor, wenn zwei parallel ausführbare Instruktionen auf eine gemeinsame Variable zugreifen, wobei mindestens ein Zugriff schreibend ist.*

In [Ber66] wird eine Herleitung der in der Definition verwendeten Kriterien gezeigt. Ebenfalls in [Ber66] wird allerdings schon angemerkt, dass diese Definition auch degenerierte Situationen umfasst, in denen ein data race keine Wettlaufsituation zur Folge hat. Schreiben zum Beispiel zwei parallele Instruktionen beide den gleichen Wert in eine Variable, so ist der Zustand des Programms unabhängig von der Ausführungsreihenfolge der beiden Instruktionen gleich. Ein ähnlicher Fall liegt vor, wenn eine Instruktion eine Variable liest und eine zweite Instruktion den Wert schreibt, der bereits in der Variablen gespeichert ist. Diese und weitere harmlose data races werden ausführlich in [NWT+07] untersucht.
Data races sind ein grundlegendes Mittel für die Implementierung von Synchronisationsfunktionen. Im einfachsten Fall liest ein Thread eine Variable v in einer Schleife solange aus, bis die Variable einen Wert ungleich 0 hat. Für diese als aktives Warten (busy-waiting) bezeichnete Funktionalität ist in Listing 1 eine mögliche Implementierung auf Assemblerebene gezeigt. Wenn ein paralleler Thread v mit dem Wert 1 beschreibt, so existiert zwischen beiden Threads ein data race bezüglich v, denn die Instruktionen auf Zeile 2 erfüllen die Kriterien von Definition 6. Wird der Programmzustand direkt nach dem Lesen von v (nach Zeile 2) betrachtet, so verursacht das data race auch eine Wettlaufsituation, denn der nach $r1$ kopierte Wert kann sowohl 0 als auch 1 sein. Der Programmzustand ist in diesem Fall also nicht deterministisch. Wird jedoch die gesamte Schleife als ein Arbeitsablauf betrachtet, so verschwindet die Wettlaufsituation. Nachdem der so definierte Arbeitsablauf ausgeführt und die Schleife verlassen wurde, sind sowohl v als auch das Register $r1$ determiniert 1.

Thread 1

```
1    // v ist 0
2  loop:
3    mov   v, r1
4    cmp   r1, #0
5    jz    loop
6    // v und r1 sind sicher 1
```

Thread 2

```
   mov   #1, v
```

Listing 1: Aktives Warten auf das Setzen einer Variable v

Daraus ergibt sich, dass die Existenz von Wettlaufsituationen in parallelen Programmen auch von der Perspektive abhängt. Insbesondere werden auf der Stufe der Maschineninstruktionen existierende data races oft so in Funktionen gekapselt, dass bei der Betrachtung kompletter Funktionen aus diesen data races keine Wettlaufsituationen herrühren.

Synchronisations-Races

Wird zu Listing 1 ein dritter paralleler Thread hinzugefügt, der ebenfalls v mit dem Wert 1 beschreibt, dann entsteht eine neue Art von Wettlaufsituationen. Zwar ist der Wert der Variablen nach Beendigung der Schleife immer noch determiniert 1, allerdings ist nicht mehr klar, ob der zweite oder dritte Thread die Wertänderung verursacht hat. Dieser Wettlauf zwischen dem zweiten und dritten Thread passiert nicht mehr aufgrund eines data races, sondern aufgrund der Semantik des gesamten Arbeitsablaufs. Er verursacht eine nicht-deterministische Programmausführung und ist ein Fehler, wenn ein deterministischer Programmablauf intendiert war. In [NM92b] werden solche Wettläufe in Abgrenzung zu *data races* als *general races* bezeichnet. In dieser Arbeit wird der Begriff *Synchronisations-Race* verwendet. Er wird in Abschnitt 4.1.2 formal beschrieben und genauer untersucht.
Eng verwandt mit Synchronisations-Races sind *message races* [NM92a, NBDK96]. Message races entstehen in nachrichtenbasierten Programmen, wenn zwei oder mehrere Nachrichten parallel zum selben Empfänger geschickt werden. Auf der Empfänger-Seite entsteht dann sowohl ein data race bezüglich des Nachrichteninhaltes als auch ein Synchronisation-Race, da kein eindeutiges Sende-Empfangs-Paar angegeben werden kann. Analysewerkzeuge für message races finden sich vor allem im MPI-Umfeld [Kra00, HPS+12].

Statischer Nichtdeterminismus

Parallele Programme enthalten oft intendierte nicht-deterministische Abläufe, um dynamisch auf ihre jeweiligen Laufzeitumgebungen reagieren zu können. In diesem Fall wartet ein Thread auf mehrere Kommunikationsereignisse. Die weitere Abarbeitung hängt dann vom jeweils eintreffenden Ereignis ab. Listing 2 stellt diesen Sachverhalt über zwei Variablen v und w dar. Thread 1 wartet in der Schleife, ob v oder w auf 1 gesetzt werden. Je nachdem, ob Thread 2 v oder Thread 3 w setzt, wird die entsprechende Funktion aufgerufen. Ein Synchronisations-Race existiert in diesem Fall nicht, da die zwei schreibenden Threads unterschiedliche Variablen nutzen. Die Arbeitsabläufe, auf die verzweigt wird (im Beispiel die Funktionen `handle_v` und `handle_w`), sind jedoch nebenläufig, wenn wie im Beispiel v und w parallel auf 1 gesetzt werden können. Der entstehende Nichtdeterminismus wird direkt auf der Ebene des Programmtextes behandelt. Entsprechend wird dieser Fall in der Arbeit als *statischer Nichtdeterminismus* bezeichnet.
Sind im Beispiel die Funktionen `handle_v` und `handle_w` äquivalent, so handelt es sich beim Arbeitsablauf auf Thread 1 um eine disjunkte Warte-Operation [Coo03]. Dieser Sonderfall des statischen Nichtdeterminismus wartet ebenfalls auf eine von mehreren Kommunikationsereignissen, setzt die Programmausführung jedoch unabhängig vom tatsächlich eingetretenen Ereignis fort.
Statischer Nichtdeterminismus ist kein Fehler, sondern für die Effizienz eines parallelen Pro-

Thread 1

```
1 while (true) {
2   if (v == 1)
3     handle_v();
4   if (w == 1)
5     handle_w();
6 }
```

Thread 2

```
v = 1;
```

Thread 3

```
w = 1;
```

Listing 2: Aktives Warten auf das Setzen einer von mehreren möglichen Variablen

gramms oft entscheidend, da so dynamisch auf Lastimbalancen im System reagiert werden kann. Wettlaufsituationen können entstehen, wenn die nebenläufigen Arbeitsabläufe zur Behandlung der Ereignisse in Wettlauf zueinander stehen. In [NM92b] werden solche Wettläufe zu den *general races* gezählt.

Nichtdeterminismus durch wechselseitigen Ausschluss

Eine weitere Form eines intendierten nicht-deterministischen Ablaufs stellt der wechselseitige Ausschluss dar [Dij65]. Wechselseitiger Ausschluss bezeichnet eine Programmsynchronisation mit Hilfe von kritischen Programmabschnitten (critical section [Ray12]). Einen bestimmten kritischen Abschnitt darf zu jedem Zeitpunkt nur maximal ein Thread ausführen. Ein in einem kritischen Abschnitt befindlicher Thread muss diesen erst wieder verlassen, bevor ein anderer Thread den Abschnitt betreten kann. Damit sind Zustandsänderungen, die durch Arbeitsabläufe im kritischen Abschnitt vorgenommen werden, für andere Arbeitsabläufe im selben kritischen Abschnitt als unteilbare Einheit sichtbar.
Die Reihenfolge der Ausführung von kritischen Abschnitten ist nicht deterministisch. Eine Wettlaufsituation entsteht jedoch wie im Fall des statischen Nichtdeterminismus nur dann, wenn der Programmzustand von der Reihenfolge der Abarbeitung dieser Abschnitte abhängt.

2.4 Wettlaufsituationen als Programmfehler

Neben intendierten Wettlaufsituationen gibt es Wettlaufsituationen, die zu fehlerhaften Programmresultaten führen können. Solche unbeabsichtigten Wettlaufsituationen sind aufgrund ihrer nicht-deterministischen Natur schwer auffindbar. Erstens ist nicht garantiert, dass der Wettlauf in einer konkreten Programmausführung zu einem Fehler führt (irreproducibility effect [SH88, NM92a]). Der Fehler kann aufgrund der nicht-deterministischen Ablaufordnung selbst auf der gleichen Maschine bei einem weiteren Lauf nicht auftreten. Noch schwieriger wird es, wenn der Fehler nur in der Produktionsumgebung auftritt, sich aber aufgrund geänderter Rahmenbedingungen in Testumgebungen nicht zeigt. Zweitens ist eine fehlerhafte Wettlaufsituation oft nicht sofort als Fehler ersichtlich. Der Fehler kann sich auf vielfältige Weise erst während der weiteren Programmausführung manifestieren. Eine direkte Verbindung vom Symptom zur Ursache ist für den Programm-Analysten dann nur schwer erkennbar. Im schlimmsten Fall wird ein falsches Programmergebnis nicht als fehlerhaft erkannt.
Fehlerhafte Wettlaufsituationen können schwerwiegende Folgen haben, da sie lange Zeit in Pro-

duktionssystemen latent vorhanden sein können, bevor ihre Wirkung überraschend sichtbar wird. Ein klassisches Beispiel ist der Therac-25-Fehler [LT93]. Der Therac-25 war ein in der Strahlentherapie eingesetzter Linearbeschleuniger. Bedingt durch eine fehlerhafte Programmierung wurden innerhalb von zwei Jahren mehrere Patienten mit zum Teil tödlichen Folgen verstrahlt. An diesem Unfall wird die Nichtreproduzierbarkeit von Wettlaufsituationen besonders deutlich. Obwohl nach einigen Vorfällen das Gerät und die Software in einer Testumgebung eingehend untersucht wurde, konnte der kritische Wettlauf nicht gefunden werden. Aufgrund veränderter Umgebungsparameter trat der Fehler in der Testumgebung nicht auf. Auch gab es eine ganze Reihe von baugleichen Apparaten, die – zumindest nach heutigem Wissensstand – immer einwandfrei funktionierten. In der Folge kam es noch zu drei weiteren Unfällen. Bei einer daraufhin erfolgten erneuten Untersuchung – diesmal unter Einsatzbedingungen – konnte der Fehler schließlich gefunden werden.

Ein weiteres Beispiel für die potentiell fatalen Folgen einer fehlerhaften Wettlaufsituation ist der flächendeckende Stromausfall im Nordosten Nordamerikas am 14.August 2003. Ein Grund für diesen Stromausfall war eine Wettlaufsituation im Energiemanagementsystem des Stromnetzes, wodurch eingehende Gefahrenmeldungen mehr als eine Stunde lang den Kontrollraum nicht erreichten [Cou04]. In der Folge kam es zu einer kaskadierenden Abschaltung von Kraftwerken, was schließlich dazu führte, dass zwei Tage lang dicht besiedelte Gebiete an der Ostküste der USA und in Ontario, Kanada ohne Strom waren. Züge fuhren nicht mehr und in einigen Regionen fiel die Trinkwasserversorgung aus. Da auch Notstromversorgungseinrichtungen nicht wie geplant funktionierten, waren auch weite Teile der sonstigen Infrastruktur betroffen.

Das zuverlässige Finden fehlerhafter Wettlaufsituationen wird damit zu einer wichtigen Aufgabe der Analyse paralleler Programme. Dabei muss das prinzipiell nicht-deterministische Verhalten solcher Programme in Betracht gezogen werden. Einzelne Testläufe sind nicht ausreichend, um die Zuverlässigkeit einer entsprechenden Analyse zu gewährleisten. Stattdessen müssen die logischen Beziehungen der Programmfunktionen innerhalb der parallelen Arbeitsabläufe untersucht werden. Dazu sind Techniken notwendig, die über das klassische Debugging der Zustandstransformationen eines Programmlaufs hinausgehen.

2.5 Reproduzierbarkeit der Analyse paralleler Programme

Sequentielle nicht-randomisierte Programme sind deterministisch. Ein bestimmter Eingabezustand wird von einem solchen Programm also immer in denselben Ausgabezustand transformiert werden. Ebenfalls determiniert sind alle Programmzustände, die sich während der Ausführung ergeben. Für die Untersuchung sequentieller Programme ist es deswegen meist ausreichend, repräsentative Eingabezustände auszuwählen und die Transformationsschritte bis zum Ausgabezustand zu analysieren. Aus einem gut gewählten Eingabezustand kann oft auch auf das Programmverhalten für andere Eingabezustände geschlossen werden. Die Untersuchung eines solchen Eingabezustandes wird auch als Test bezeichnet.

Definition 7. *Ein* Test *ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteins relativ zu vorher festgelegten Anforderungen [DS91].*

Diese Definition erhebt im Gegensatz zu anderen Varianten [PKS02, iee90] explizit die Forderung nach Reproduzierbarkeit. Um dieser Forderung auch für parallele Programme gerecht werden zu können, muss jedoch eine weitere Untersuchungsdimension hinzugefügt werden, da für jeden Eingabezustand verschiedene Varianten der Programmausführung existieren. Ein Test kann nicht nur die Zustands-Transformationen einer konkreten Programmausführung betrachten, sondern muss auch untersuchen, ob das Ergebnis von möglichen Ablauf-Varianten abhängt.
Wie in Abschnitt 2.3 erläutert wurde, können parallele Programme im Gegensatz zu sequentiellen Programmen nicht-deterministische Abläufe enthalten. Die Schritte, die einen Eingabezustand in einen Ausgabezustand transformieren, können in jedem Programmlauf in einer anderen zeitlichen Reihenfolge stattfinden. Auch eine Permutation des Ablaufs, die über eine lediglich zeitliche Verschiebung von Programmereignissen hinausgeht, ist möglich. Für eine Programmanalyse ist es also nicht mehr ausreichend, eine Menge von Eingabe- und Ausgabezuständen zu betrachten. Ebenfalls betrachtet werden müssen die für jeden untersuchten Eingabezustand möglichen Permutationen der Programmausführung.
Eine Lösung für diese Herausforderung ist die Analyse der Aufzeichnung einer Programmausführung. Diese Methode besteht aus zwei prinzipiellen Schritten:

1. Aufzeichnung eines Programmlaufs

2. Analyse der aufgezeichneten Programmereignisse.

Viele Analysetechniken für parallele Programme nutzen diese Methode, da sie es erlaubt, den kompletten Programmablauf in die Untersuchung mit einzubeziehen. Die *record & replay*-Technik simuliert äquivalente Programmläufe anhand der Aufzeichnung und schafft so eine kontrollierte Testumgebung [LSZ90]. Eine Variante dieser Technik modifiziert die aufgezeichneten Ereignisse, um zum Beispiel alternative Abarbeitungsreihenfolgen testen zu können [GV95, Kra00]. Das Konzept des *Active Testing* [Sen08, JNPS09] erzeugt alternative Abarbeitungsreihenfolgen durch die gezielte Verzögerung bestimmter Programmereignisse.
Andere Verfahren verzichten auf ein *replay* im Analyseschritt und beweisen stattdessen anhand der aufgezeichneten Ereignisse bestimmte Programmeigenschaften auch für alternative Abarbeitungsreihenfolgen [NBDK96, RM95].
Die eben genannten Verfahren sind *post-mortem*-Analysen. Post-mortem-Analysen speichern die während der Aufzeichnung gewonnenen Daten. Der eigentliche Analyse-Schritt findet dann nach der Beendigung des Programms statt. Im Gegensatz zum post-mortem-Verfahren verzichten *on-the-fly*-Techniken [GZH+94] auf eine dedizierte Speicherung des Programmlaufs. Die Analyse stattgefundener Programmereignisse findet schon während der Ausführung statt. Damit wenden letztendlich auch on-the-fly-Techniken die beiden oben aufgeführten Schritte an. Allerdings werden beide Schritte verschränkt ausgeführt, so dass sie für den Programm-Analysten nicht mehr sichtbar zu trennen sind. Ein Vorteil von *on-the-fly*-Techniken ist die Möglichkeit, dass die Ausführung des analysierten Programms von Analyseergebnissen direkt beeinflusst werden kann [SH11].
Ein anderer Lösungsansatz für das Problem der Reproduzierbarkeit ist die symbolische Ausführung (*symbolic execution* [BCO05]). Darauf aufbauende Werkzeuge erstellen ein System aus

Constraints über den Programmzustand, über dieses dann Programmeigenschaften bewiesen werden können [BDDP11].

2.6 Task Graphen

Allen im vorherigen Abschnitt vorgestellten Verfahren ist gemein, dass sie im Analyseschritt auf einem Modell der Programmausführung operieren. Die Ausführung wird in einzelne Programmereignisse zerlegt, die dann untersucht werden. In Graphenmodellen stellen diese Programmereignisse einzelne Berechnungsschritte des Programms dar und werden als Knoten im Graphen repräsentiert [Sin07]. Die Kanten eines solchen Graphen sind meist gerichtet und werden zur Modellierung verschiedener Eigenschaften eingesetzt.
In einem Programmabhängigkeitsgraphen stellen die Kanten kausale Abhängigkeiten zwischen einzelnen Berechnungsschritten dar. Sind zwei Knoten A und B durch einen Pfad $A \rightarrow B$ miteinander verbunden, so muss erst A ausgeführt werden, bevor B ausgeführt werden kann. Zulässige Programme haben immer azyklische Programmabhängigkeitsgraphen.
In einem Kontrollflussgraph modellieren die Kanten mögliche Übergänge von einem Berechnungsschritt zum nächsten. Ein Pfad durch einen Kontrollflussgraph modelliert also einen möglichen Programmablauf. Im Gegensatz zu Programmabhängigkeitsgraphen können Kontrollflussgraphen Zyklen – z. B. für die Modellierung von Schleifen – enthalten.
In dem in [Sin07] definierten Task Graphen repräsentieren gewichtete Kanten die Kommunikation zwischen einzelnen Berechnungsschritten. Unter der dort ebenfalls getroffenen Annahme, dass eine Kommunikationsbeziehung auch eine Abhängigkeit etabliert, besteht die Überführung eines Task Graphen in einen Programmabhängigkeitsgraphen nur aus der Entfernung der Kantenwichtungen und ist somit trivial.
In der vorliegenden Arbeit wird der Begriff des Task Graphen für die Modellierung einer Programmausführung verwendet. Eine Programmausführung bezeichnet die Befehlsabfolge eines konkreten Programmlaufs. Ein solcher Task Graph ist demzufolge nur zur dynamischen Programmanalyse geeignet, da nicht ausgeführte Programmpfade im Task Graphen nicht repräsentiert sind. Berechnungsschritte werden zu Ereignissen innerhalb einer Programmausführung abstrahiert.
Der Begriff *Task* ist selbst dann mehrdeutig, wenn er nur im Kontext der Informatik betrachtet wird. In [IBM17] werden insgesamt 31 verschiedene Bedeutungen aufgelistet. In dieser Arbeit wird der Begriff wie folgt definiert:

Definition 8. *Ein* Task *ist ein Modell für ein abgeschlossenes Ereignis in einer Programmausführung zu einem bestimmten Zeitpunkt.*

Die gesamte Ausführung eines Programms lässt sich somit in einzelne Tasks zerlegen, wobei jeder Task genau einmal ausgeführt wird. Innerhalb eines Tasks wird keine weitere Zerlegung vorgenommen und die Reihenfolge der Ausführung der in ihm zusammengefassten Berechnungen ist undefiniert. Tasks werden somit logisch atomar, also zu genau einem bestimmten Zeitpunkt, ausgeführt. Weiterhin sind Tasks abgeschlossen: bei Eintritt in einen Task müssen alle benötigten Eingabedaten vorliegen und nach Verlassen des Tasks sind alle Ausgabedaten sichtbar.

Ein Task ist nicht notwendigerweise äquivalent zu einer einzelnen Anweisung. Oft ist es sinnvoll, mehrere Anweisungen oder auch die Ausführung einer Funktion zu einem Task zusammenzufassen. Andererseits kann ein Task auch ein rein logisches Ereignis, z. B. das Betreten oder Verlassen einer Funktion, repräsentieren. Darüber hinaus ist es unter Umständen sogar notwendig, für eine präzise Modellierung einzelne Maschinenbefehle in mehrere Tasks zu zerlegen. Beschreibt zum Beispiel ein Maschinenbefehl eine bestimmte Speicherzelle, so muss aufgrund der out-of-order Ausführung von Befehlen in modernen Mikroprozessoren dieser Befehl in die zwei Tasks *Instruktionsausführung* und *Schreiben des Wertes* zerlegt werden, um die Eigenschaft der Abgeschlossenheit der einzelnen Tasks zu wahren. In Abschnitt 4.2.2 wird die Zerlegung von asynchronen Kommunikationsoperationen in mehrere Tasks näher beschrieben.
Die einzelnen Tasks einer Programmausführung können mittels ihrer kausalen Relationen zueinander in Beziehung gesetzt werden. Für diese Relation wurde der Begriff *happened-before-Relation* geprägt [Lam78].

Definition 9. *Eine* happened-before-Relation *($\prec$) bezeichnet eine Relation zwischen zwei Tasks A und B. Es gilt $A \prec B$ genau dann, wenn B nur nach A ausgeführt werden kann.*

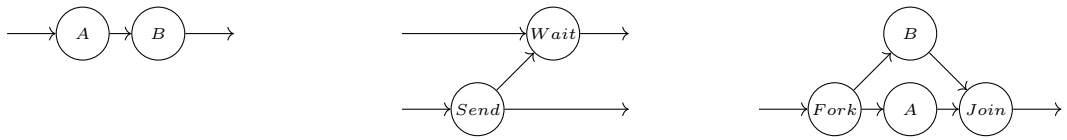
Die happened-before-Relation ist eine strenge Ordnungsrelation und demnach

irreflexiv $A \not\prec A$,

antisymmetrisch $A \prec B \Rightarrow B \not\prec A$,

transitiv $A \prec B \wedge B \prec C \Rightarrow A \prec C$.

Die einfachste Form einer happened-before-Relation besteht zwischen nacheinander abgearbeiteten Tasks eines Threads. Jeder Task kann nur ausgeführt werden, nachdem der direkt davor liegende Task ausgeführt wurde.
In parallelen Programmausführungen treten zusätzliche happened-before-Relationen zwischen Tasks unterschiedlicher Threads auf. Sendet der Task A eines Threads eine Nachricht und wartet ein Task B eines anderen Threads auf genau diese Nachricht, so gilt $A \prec B$. Darüber hinaus entstehen happened-before-Relationen, wenn ein Task einen neuen Thread startet bzw. wenn ein Task auf den Abschluss eines Threads wartet. Abbildung 2.1 illustriert mögliche happened-before-Relationen.



Tasks in einem Thread

Tasks in einer Sende/Warte-Beziehung

Tasks in einer Fork/Join-Beziehung

Abbildung 2.1: Mögliche Ausprägungen der happened-before-Relation

Datenabhängigkeit induziert eine weitere Form der happened-before-Relation. Allgemein muss ein zu schreibender Wert erst ermittelt werden. Wird zum Beispiel ein Wert von einer Speicherzelle in eine andere Speicherzelle kopiert, so kann die Schreiboperation nur nach der Leseoperation ausgeführt werden.
Die Menge aller Tasks einer Programmausführung wird zusammen mit der transitiven Reduktion der happened-before-Relationen in einem Task Graphen zusammengefasst.

Definition 10. *Ein* Task Graph *ist ein azyklischer, gerichteter Graph $\mathcal{P} = \langle E, \prec \rangle$, der die happened-before-Relationen zwischen den Tasks einer Programmausführung $\mathcal{P}$ modelliert. Die endliche Menge E enthält alle ausgeführten Tasks und $\prec$ ist die happened-before-Relation über E.*

Ein Task Graph repräsentiert somit alle logisch möglichen Abarbeitungsreihenfolgen der einzelnen Tasks einer Programmausführung. Existiert in einem Task Graphen ein Pfad von einem Task A zu einem Task B, so gilt aufgrund der Transitivität der happened-before-Relation $A \prec B$. Somit existiert eine garantierte Abarbeitungsreihenfolge von A und B, da B laut Definition 9 nur nach A ausgeführt werden kann. Existiert weder ein Pfad von A nach B noch ein Pfad von B nach A, so ist die Reihenfolge der Ausführung von A und B undefiniert. Dann können beide Tasks in beliebiger Reihenfolge und praktisch auch gleichzeitig ausgeführt werden.
Mit Hilfe eines Task Graphen sind data races also auch dann feststellbar, wenn diese in der aufgezeichneten Ausführung zu keinem Fehler geführt haben. Dazu müssen Speicherzugriffe als Programmereignisse erfasst werden. Dann können die Kriterien von Definition 6 zur Überprüfung dieser Zugriffsereignisse verwendet werden. Ob zwei Ereignisse parallel ausführbar sind, ergibt sich dabei aus dem Task Graphen.
Daraus resultiert die Aufgabe, zu einer gegebenen Programmaufzeichnung einen Task Graphen zu erstellen. Für die in dieser Arbeit behandelten PGAS-Programme ergibt sich zusätzlich die Forderung, dass die durch PGAS-Operationen ausgelösten Speicherzugriffe im Task Graph repräsentiert werden müssen. Dabei sollen auch die kausalen Beziehungen zu den Tasks der Threads im Graph abgebildet werden. Eine weitere Aufgabe ist das Finden von Synchronisations-Races während der Erstellung eines Task Graphen. Kann gezeigt werden, dass eine Programmaufzeichnung eindeutig in einen Task Graphen überführt werden kann, so kann damit auch die Existenz von Synchronisations-Races für die untersuchte Programmausführung ausgeschlossen werden.

2.7 GASPI – Global Address Space Programming Interface

Die in dieser Arbeit entwickelten Werkzeuge analysieren verteilte Anwendungen, die mit GASPI als Kommunikationssubstrat entwickelt wurden. GASPI ist eine Programmierschnittstelle für Operationen in einem PGAS-System. Diese Programmierschnittstelle wurde im gleichnamigen BMBF-Projekt von 2011 bis 2013 entwickelt [ABB+13] und wird seitdem vom GASPI-Forum gepflegt [GAS11].

Ausführungsmodell

Ein GASPI-Programm besteht aus einer Anzahl von Prozessen, wobei jedem Prozess ein eindeutiger *Rank* zugeordnet ist. Ranks bezeichnen eine aufsteigende Nummerierung der Prozesse, beginnend mit Rank 0. Sie werden den Prozessen statisch beim Start zugewiesen. Die Prozesse werden unabhängig voneinander vom Betriebssystem gestartet und kommunizieren dann mittels GASPI-Funktionen miteinander. Ein SPMD-Programmiermodell wird von GASPI nicht vorausgesetzt.

Ein Prozess kann aus mehreren Threads bestehen. Anders als in MPI gibt es keine verschiedenen Thread-Umgebungen. Alle Threads können gleichberechtigt alle GASPI-Funktionen aufrufen. Allerdings bietet GASPI selbst keine Programmierschnittstelle für Multi-Threading. Hierzu muss der Programmierer auf andere Schnittstellen (z. B. OpenMP) zurückgreifen.

Bevor ein Prozess GASPI-Operationen durchführen kann, muss `gaspi_proc_init` als erste Funktion aufgerufen werden. Standardmäßig baut diese Funktion die Kommunikationsinfrastruktur zwischen dem aufrufenden Prozess und allen entfernten GASPI-Prozessen auf. Dieses Vorgehen kann auf großen Systemen ineffizient werden. Deswegen kann der Programmierer den Aufbau der Kommunikationsinfrastruktur in `gaspi_proc_init` auch unterdrücken und diese später nur zwischen bestimmten Ranks etablieren.

Speichermodell

Das GASPI-Speichermodell lehnt sich an das in Abschnitt 1.1 bereits beschriebene PGAS-Speichermodell an. Jeder Prozess besitzt neben seinem lokalen Speicherbereich auch eine oder mehrere Partitionen des gemeinsamen Speicherbereichs. Diese Partitionen werden in GASPI als *Segmente* bezeichnet. Ein von einem Prozess alloziertes Segment wird zur lokalen Partition für diesen Prozess. Jedem Segment wird bei der Allokation vom Programmierer eine innerhalb des Prozesses eindeutige Nummer zugewiesen. Ein Segment wird also durch die Segmentnummer und den Rank des das Segment beherbergenden Prozesses bezeichnet. Ein Segment fungiert lokal wie ein Heap-Speicher, in dem Variablen abgelegt werden können. Dazu kann der lokale Prozess die Funktion `gaspi_segment_ptr` aufrufen, die einen Zeiger auf das erste Byte eines Segments liefert. Auf entfernte Segmente kann ein Prozess nur über explizite put- und get-Operationen zugreifen. Die Adresse einer Variable in einem Segment wird durch den Offset zum Start des Segments bestimmt. Die Position einer Variable A_R im gemeinsamen Speicherbereich wird also eindeutig durch das Tripel $A_R = \{rank, segment, offset\}$ definiert. Befindet sich eine Variable in einem lokal allozierten Segment, dann ist ihre Adresse A_L durch das Tupel $A_L = \{segment, offset\}$ bereits ausreichend definiert.

Prozesse können Segmente unabhängig voneinander erstellen. Dadurch werden heterogene Systeme unterstützt, in denen Knoten unterschiedliche Speicherarchitekturen besitzen. Des Weiteren ist es für einen Prozess möglich, mehrere Segmente zu erstellen. Abhängig vom System können diese in unterschiedlichen Speichertypen (z. B. NVRAM, High Bandwidth Memory) alloziert werden. Auf diese Weise kann der Programmierer die jeweiligen Speichereigenschaften gezielt für jeweils geeignete Aufgaben innerhalb des PGAS-Programmiermodells nutzen.

RMA-Operationen

Alle von GASPI definierten RDMA-Operationen sind asynchrone Operationen. Sie kehren also schon zurück, noch bevor der Lese- bzw. Schreibvorgang auf dem lokalen Speichersegment abgeschlossen ist. RDMA-Operationen werden von GASPI in Queues organisiert. Queues sind prozess-lokale Entitäten, die die Abarbeitung der gestarteten Operationen gewährleisten. Sie sind aufsteigend nummeriert und werden vom Programmierer direkt angesprochen.
Die prinzipielle Funktion zum Starten einer put-Operation ist `gaspi_write(`Q, A_L, A_R, S`)`. Diese Operation kopiert asynchron S Bytes von der lokalen Quelladresse A_L zur Zieladresse A_R unter Verwendung von Queue Q. Die entsprechend fundamentale Funktion zum Starten einer get-Operation ist `gaspi_read(`Q, A_R, A_L, S`)`, welche asynchron S Bytes von der entfernten Quelladresse A_R zur lokalen Zieladresse A_L transferiert.
Die Funktion `gaspi_wait(`Q`)` wartet auf den Abschluss aller asynchronen lokalen Lese- und Schreibvorgänge, die in der Queue Q platziert sind. Kehrt diese Funktion zurück, ist jedoch nur für `gaspi_read` auch der Abschluss des entfernten Lesezugriffs garantiert. Für `gaspi_write` existiert keine Zusicherung, dass auch der entfernte Schreibzugriff beendet ist. Auf den Abschluss dieses Schreibzugriffs muss der Zielprozess testen. Diese Synchronisation geschieht unabhängig vom auslösenden Prozess.

Synchronisationsmodell

GASPI definiert kollektive und gerichtete Synchronisationsfunktionen. Kollektive Synchronisationsfunktionen sind Barrieren, die die Ausführung einer Gruppe von Prozessen so lange unterbrechen, bis alle Prozesse dieser Gruppe die Barriere betreten haben. Barrieren können nicht-blockierend sein. Damit können Prozesse zwischen dem Betreten der Barriere und dem Warten auf alle anderen Prozesse der Gruppe weitere Arbeit verrichten. Für die Gesamtheit aller GASPI-Prozesse in einem Programm existiert die vordefinierte Gruppe `GASPI_GROUP_ALL`.
Gerichtete Synchronisation wird durch einen Notifikationsmechanismus implementiert. Prozesse können kurze Nachrichten – sogenannte Notifikationen – an andere Prozesse schicken. Jedes Segment eines Prozesses besitzt einen privaten Speicherbereich, in den andere Prozesse Notifikationen schreiben können. Die Adresse einer Notifikation ist bestimmt durch den Zielprozess, das Zielsegment und einen Offset innerhalb des privaten Speicherbereichs. Der Offset wird auch als Notifikations-ID bezeichnet, das Tripel $A_N = \{rank, segment, notification\text{-}id\}$ definiert also die Adresse einer Notifikation. Die Funktion `gaspi_notify(`Q, A_N, V`)` sendet einen integralen Wert V ungleich 0 asynchron über die Queue Q an die Notifikations-Adresse A_N. GASPI garantiert, dass zuvor über dieselbe Queue Q zum selben Zielprozess ausgeführte put-Operationen (`gaspi_write`) auf dem Zielknoten abgeschlossen sind, bevor die Notifikation dort sichtbar wird.
Dem Zielprozess stehen zwei Funktionen zum Testen von eingehenden Notifikationen zur Verfügung. Die Funktion `gaspi_waitsome(`S, ID_{Start}, ID_{End}`)` wartet darauf, dass in einem lokalen Segment S mindestens eine Notifikation innerhalb des fortlaufenden Bereichs von Notifikations-IDs $[ID_{Start}, ID_{End})$ auf einen Wert ungleich 0 gesetzt wird. Ist bereits eine Notifikation gesetzt, so kehrt die Funktion sofort zurück. Die Funktion `gaspi_reset(`S, ID`)` setzt eine Notifikation an Offset ID auf 0 zurück und gibt den vorher dort gespeichert Wert zurück. Anhand des Rück-

gabewertes entscheidet der GASPI-Prozess dann über die weitere Abarbeitung. `gaspi_reset` arbeitet atomar. Damit ist es möglich, dass mehrere Threads ohne zusätzliche Synchronisation gleichzeitig auf die gleiche Notifikation warten können.
Die Synchronisation geschieht in GASPI immer mittels `gaspi_reset`. Zwar wird auch von `gaspi_waitsome` die Notifikations-ID zurückgegeben, welche die Beendigung des Warte-Zustandes verursacht hat. Allerdings handelt es sich dabei nur um einen Hinweis zur Vereinfachung eines nachfolgend erforderlichen `gaspi_reset`-Aufrufs. Eine zuverlässige Feststellung des Wertes einer Notifikation ist nur durch `gaspi_reset` möglich. Eine GASPI-Implementierung kann in `gaspi_waitsome` eine plattformabhängige effiziente Warte-Strategie bereitstellen. Ein Programm kann jedoch auch auf eine Verwendung von `gaspi_waitsome` verzichten und stattdessen den Zustand von Notifikationen wiederholt mit `gaspi_reset` abfragen und in der Zwischenzeit andere Arbeit verrichten.
In der Praxis werden `gaspi_waitsome` und `gaspi_reset` häufig zusammen verwendet. Listing 3 zeigt die Implementierung der Funktion `gaspi_wait_reset`, die aus diesen beiden Operation zusammengesetzt ist. Diese Funktion wartet im Segment S auf eine Notifikation aus dem Bereich $[ID_{Start}, ID_{Start} + N)$ und gibt die Notifikations-ID zurück, die die Beendigung des Wartens verursacht hat. Durch die Auswertung der zurückgegebenen ID modelliert ein GASPI-Programm statischen Nichtdeterminismus. Die Funktion `gaspi_wait_reset` wird im weiteren Verlauf der Arbeit zur Vereinfachung in GASPI-Listings verwendet.

```
1 function gaspi_wait_reset(S, ID_Start, N)
2 {
3   flag_value = 0;
4   while (flag_value == 0) {
5     ID = gaspi_waitsome(S, ID_Start, ID_Start + N);
6     flag_value = gaspi_reset(S, ID);
7   }
8   return ID;
9 }
```

Listing 3: Die `gaspi_wait_reset`-Funktion als Kombination der zwei Basisfunktionen `gaspi_waitsome` und `gaspi_reset`

3 Speicherzugriffsanalyse paralleler Programme – Stand der Forschung

> *Program testing can be used to show the presence of bugs, but never to show their absence! [Dij72]*

Die Bedeutung des PGAS-Programmiermodells ist im letzten Jahrzehnt stetig gewachsen. Neuere Arbeiten [MRH14, SWS+12] deuten darauf hin, dass dieses Programmiermodell den Herausforderungen moderner HPC-Architekturen gerecht wird. Trotz dieser gewachsenen Bedeutung ist das Arsenal an Analysewerkzeugen zur Untersuchung speziell von PGAS-Programmen noch sehr überschaubar.
Dieses Kapitel vermittelt einen Überblick über Methoden und Werkzeuge, die mit den in der Arbeit behandelten Themen in Zusammenhang stehen.

3.1 Die Ermittlung von Task Graphen und Synchronisations-Races

Wie in Abschnitt 2.6 erläutert wurde, ist die Modellierung von Beziehungen zwischen Threads eine Grundlage für eine weiterführende Analyse paralleler Programme. Eine fundamentale Beziehung ist dabei die happened-before-Relation. Zur Darstellung der happened-before-Relation wird in [EGP89] ein Synchronisationsmodell definiert, in dem das *Flag* das grundlegende Konzept zur Kommunikation zwischen Threads bildet. Ein Flag hat zwei mögliche Zustände: *gesetzt* und *gelöscht*. Auf einem Flag werden drei Basisoperationen definiert:

- *POST* ändert den Zustand des Flags auf *gesetzt*.
- *WAIT* unterbricht die weitere Ausführung des Threads solange, bis der Zustand des Flags *gesetzt* ist. Wenn das Flag bei Eintritt in die *WAIT*-Operation bereits den Zustand *gesetzt* hat, wird die Ausführung des Threads direkt fortgesetzt.
- *CLEAR* ändert den Zustand des Flags auf *gelöscht*.

Für die Berechnung eines Task Graphen zu einer gegebenen Programmausführung bestehend aus diesen drei Operationen wird der *common-ancestor* Algorithmus angegeben. Dieser Algorithmus arbeitet in polynomialer Zeit bezogen auf die Anzahl der Synchronisationsoperationen, findet allerdings unter Umständen nicht alle happened-before-Relationen. Aus diesem Grund wird die Arbeit in [EGP92] um den *exhaustive-pairing* Algorithmus erweitert. Mit diesem Algorithmus werden zu einer Programmausführung alle möglichen Task Graphen vollständig ermittelt. Allerdings hat der Algorithmus exponentielle Komplexität.
In [NM90] wird bewiesen, dass das Problem der Berechnung eines Task Graphen zu einer gegebenen Programmausführung bestehend aus den drei genannten Basisoperationen NP-vollständig

ist. Der Beweis besteht aus zwei Schritten. Im ersten Schritt wird ein Programm bestehend aus kritischen Programmabschnitten konstruiert, für das die Berechnung eines Task Graphen auf das 3-SAT-Problem zurückgeführt wird. Im zweiten Schritt wird gezeigt, dass eine eingeschränkte Form kritischer Programmabschnitte mit Hilfe der drei Basisoperationen *POST*, *WAIT* und *CLEAR* programmiert werden kann. Dazu wird vor einem *WAIT* auf ein Flag ein *CLEAR* auf ein anderes Flag ausgeführt. Diese Form wird dann wieder auf das 3-SAT-Problem zurückgeführt.

3.1.1 Synchronisationsbeziehungen in MPI

Eng verwandt mit der Berechnung eines Task Graphen zu einer gegebenen Programmausführung ist die Ermittlung von Synchronisationsbeziehungen zwischen Send- und Receive-Operationen in MPI-Programmen. In ScoreP/OTF2 wird die Reihenfolge der aufgezeichneten Ereignisse genutzt, um Verbindungen zwischen Send-Operationen und dazu passenden Receive-Operationen herzustellen [OTF15]. Das Vorgehen in [Kra00] setzt voraus, dass während der Aufzeichnung zu jeder Operation der bzw. die Partnerprozesse wechselseitig bestimmt werden können. Anhand dieser Partnerbeziehungen und der Reihenfolge der Ereignisse wird dann ein Task Graph erstellt. MUST [HPS+12] zeichnet die zur Laufzeit entstehenden Beziehungen zwischen MPI-Operationen mit auf. Der Task Graph wird direkt aus der Aufzeichnung erstellt. Mit der in [HSN+13] beschriebenen Erweiterung können auch Wildcard-Operationen und kollektive Synchronisationen behandelt werden. Die Daten werden während des Programmlaufs zur Diagnose von Deadlocks eingesetzt. Mit diesen Methoden wird zu einem gegebenen Lauf eines MPI-Programms genau ein Task Graph erstellt. Es wird nicht festgestellt, ob der erstellte Task Graph eindeutig ist oder ob eine Programmausführung mit einer nur zeitlichen Verschiebung der Ereignisse zu einem anderen Task Graphen führen würde.
In [VGK+11] werden die kausalen Zusammenhänge zwischen MPI-Operationen ermittelt. Dazu wird die happened-before-Relation durch eine MPI-spezifische *matched-before-Relation* ersetzt. Mit Hilfe dieser Relation werden MPI-Operationen logische Zeitstempel zugeordnet. Mit diesen Zeitstempeln können dann sowohl kausale als auch nichtdeterministische Synchronisationsbeziehungen in einer Programmausführung ermittelt werden.
Ein Spezialfall nichtdeterministischer Synchronisationsbeziehungen in MPI-Programmen sind *message races*. Wie Synchronisations-Races führen *message races* aufgrund der Kopplung von Synchronisation und Nachrichtenübertragung zu einer nichtdeterministischen Ordnung der Programmereignisse innerhalb eines Programmlaufs. Aufgrund der Semantik der nachrichtenbasierten Synchronisation ist die Ermittlung von *message races* nicht mehr NP-vollständig. Eine erste Untersuchung von *message races* wird in [NM92a] durchgeführt. In [NBDK96] wird der Ansatz erweitert und *message races* werden sowohl lokalisiert als auch klassifiziert. Dabei sind ursprüngliche (*nonartifact*) *message races* solche Races, welche nicht durch vorhergehende Races verursacht werden. Das Verfahren basiert auf der Untersuchung der von den Send- und Receive-Operationen erzeugten Task Graphen. Die Lokalisierung von *message races* geschieht in einer on-the-fly Analyse. Da diese Analyse jedoch eine potentiell unüberschaubare Menge an Races ausgeben kann, werden in einem zweiten post-mortem Schritt die ursprünglichen *message races* ermittelt. Nur diese Races werden ausgegeben.

Der in [Kra00] beschriebene Ansatz stellt für blockierende und nicht-blockierende *receive*-Operationen im generierten Task Graphen fest, ob zusätzliche passende parallele *send*-Operationen existieren. Solche Operationen werden als *message races* markiert.

3.1.2 Die Modellierung von Synchronisationsbeziehungen in PGAS-Systemen

Das Problem der Modellierung von PGAS-Programmen wird in [CDMM13] behandelt. Aus PGAS-Operationen wird ein spezieller Task Graph erstellt, der auch asynchron ausgelöste Speicherzugriffe abbildet. Wird in diesem Task Graph ein Zyklus gefunden, so enthält das zugrundeliegende Programm Fehler in der Speicherzugriffskoordination. Der Ansatz kann jedoch nicht alle Fehler finden, da zyklische Abhängigkeiten in dem entwickelten Modell zwar ein hinreichendes, nicht aber notwendiges Kriterium für data races sind [Der13].
Ein weiterer Ansatz zur Modellierung von PGAS-Programmen wird in [DLHV16] behandelt. Diese Arbeit beschreibt eine Sprache *coreRMA*, die grundlegende RDMA-Operationen abstrahiert. Mit dieser Sprache werden Tests erstellt, die das definierte Speichermodell gegen PGAS- und Netzwerk-Implementierungen überprüft. Die beschriebene Sprache beschränkt sich auf Speicheroperationen. Synchronisationsoperationen werden nicht betrachtet.

3.2 Speicherzugriffsanalyse

Speicherzugriffsanalyse wird in der Softwareentwicklung für verschiedene Aufgabenbereiche eingesetzt. Grundsätzlich zu unterscheiden sind Korrektheits- und Effizienzanalysen.
Im Bereich der Korrektheitsanalyse muss zwischen Fehlern, die bereits in seriellen Programmen auftreten können und solchen, die typisch für parallele Programme sind, unterschieden werden. Speicherzugriffsfehler in seriellen Programmen sind z. B.

- Pufferüberläufe,
- Lesen uninitialisierter Variablen,
- Zugriffe auf nicht allozierten Speicher.

Die Analyse solcher Fehler wird von Memory-Debuggern wie z. B. Allinea DDT [All18] oder Memcheck [SN05] unterstützt. Diese Werkzeuge erkennen fehlerhafte Speicherzugriffe auch dann, wenn diese sich nicht sofort in einem ersichtlichen Fehler manifestieren. Aufgrund des Themas dieser Arbeit konzentriert sich der folgende Überblick auf Werkzeuge, die speziell für die Speicherzugriffsanalyse paralleler Programme entwickelt wurden.

3.2.1 Korrektheitsanalyse von Speicherzugriffen in nicht-verteilten Systemen mit gemeinsam genutztem Speicher

Data races gehören zu den bekannten und berüchtigten Fehlern in der parallelen Programmierung. Demzufolge existieren eine Vielzahl von Verfahren und Werkzeugen zum Auffinden von data races. Die Entwicklung hat sich dabei auf Multithreaded-Systeme, also nicht-verteilte Systeme mit gemeinsam genutztem Speicher, konzentriert. Eine Auswahl von Werkzeugen für diese Systeme wird in Tabelle 3.1 gezeigt.

Werkzeug	Verfahren	Realisierung
Eraser [SBN+97]	LockSet-Analyse	dynamische Binärcode-Instrumentierung und on-the-fly Analyse
MultiRace [PS07]	LockSet-Analyse und Vector Clocks	C++-Compiler-Instrumentierung und on-the-fly Analyse
FastTrack [FF09]	optimierte Vector Clocks	dynamische Java-Code-Instrumentierung und on-the-fly Analyse
SPD3 [RZS+12]	Dynamic Program Structure Tree	dynamische Java-Code-Instrumentierung und on-the-fly Analyse
asyncStar [BDDP11]	Speicherzugriffsrechte in *separation logic*	statische Quellcode-Analyse

Tabelle 3.1: Werkzeuge zum Auffinden von data races in Programmen mit gemeinsam genutztem Speicher

Eraser [SBN+97] führte die LockSet-Analyse von kritischen Abschnitten ein. Vor dieser Arbeit wurden kritische Abschnitte auf happened-before-Relationen zurückgeführt. Dieses Vorgehen hatte den Nachteil, dass die kritischen Abschnitte nur in der Reihenfolge ihres tatsächlichen Auftretens analysiert wurden. Alternative Ausführungsreihenfolgen wurden nicht untersucht. Das LockSet-Verfahren zeichnet stattdessen die von einem Thread gehaltene Menge der Locks für jede Variable auf. Die Analyse bildet die Schnittmenge über alle Threads. Wenn diese Menge leer wird, gibt Eraser eine Warnung über ein eventuelles data race aus.

MultiRace [PS07] kombiniert das LockSet-Verfahren mit einer Vector-Clock-Analyse. Vector Clocks [Mat88] modellieren die happened-before-Relationen zwischen Threads. Durch die Beachtung dieser happened-before-Relationen werden Speicherzugriffe, die nicht über einen gemeinsamen kritischen Abschnitt gesichert, sondern durch Synchronisation geordnet sind, korrekt nicht als data races erkannt. MultiRace demonstriert, dass beide Verfahren die gleichen Ressourcen nutzen können und Informationen des jeweils anderen Verfahrens für eine effizientere Analyse verwendbar sind.

Ebenfalls auf der Vector-Clock-Analyse basiert FastTrack [FF09]. Der für das Werkzeug entwickelte Algorithmus verbessert die Platz- und Zeitkomplexität der Vector-Clock-Analyse. FastTrack benötigt für jede Variable unabhängig von der Anzahl der Threads nur noch konstanten Speicherplatz und für jeden Speicherzugriff nur eine konstante Zeitdauer.

In SPD3 [RZS+12] wird die Analyse mit Hilfe eines *Dynamic Program Structure Tree* vorgenommen. Diese Analyse kann Programme mit geschachtelter fork-join-Semantik verarbeiten. Das Verfahren berechnet aus den verwendeten parallelen Konstrukten einen Baum, in dem die Knoten fork- oder join-Operationen und die Blätter die Berechnungen repräsentieren. In einem solchen Baum kann sehr effizient auf parallel ausführbare Anweisungen und damit auch auf Zugriffskonflikte geschlossen werden. Es existiert eine Implementierung als Bibliothek für die dynamische Analyse von Habanero Java Programmen (*Scalable Precise Dynamic Datarace Detection*, kurz SPD3).

Das Werkzeug *asyncStar* [BDDP11] führt eine statische Programmanalyse einer C-artigen Sprache durch, die um asynchrone Speicheroperationen erweitert wurde. Das zugrundeliegende Modell berechnet die Gewährung von Lese- und Schreibrechten mit *separation logic* [BCOP05]. Lese-

und Schreibrechte für einen bestimmten Speicherbereich werden durch beliebige reelle Zahlen im Intervall $(0, 1]$ repräsentiert. Besitzt ein Thread die Berechtigung 1 für einen Speicherbereich, so hat er exklusiven Schreibzugriff. Jede Berechtigung zwischen 0 und 1 berechtigt zum Lesen. Berechtigungen werden beim Starten neuer Threads aufgeteilt und beim Zusammenführen von Threads wieder zusammenaddiert. Dadurch wird gewährleistet, dass zu jedem Zeitpunkt die Summe aller Berechtigungen für jeden Speicherbereich immer 1 ist. In [BDDP11] wird dieses Modell auf asynchrone Speicheroperationen angewendet. *AsyncStar* kann für ein gegebenes Multithreading-Programm das Nichtvorhandensein von data races zwischen Speicherzugriffen und potentiell gleichzeitig stattfindenden asynchronen Speicheroperationen beweisen.

3.2.2 Korrektheitsanalyse von Speicherzugriffen in verteilten Systemen

Verteilte Systeme mit gemeinsam genutztem Speicher wie z. B. PGAS-Systeme erlangen erst seit einigen Jahren zunehmende Bedeutung in der HPC-Programmierung. Entsprechend existieren bisher nur wenige Werkzeuge zur Speicherzugriffsanalyse auf solchen Systemen. *SyncChecker* [CLC+12] findet data races in MPI-Programmen mit asynchroner Kommunikation. Mit dem Instrumentierungs-API PIN [LCM+05] werden zur Laufzeit alle Speicherzugriffe und aufgerufenen MPI-Funktionen aufgezeichnet. In der Analysephase wird für jeden Prozess untersucht, ob während aktiver asynchroner MPI-Operationen Speicherzugriffe auf Adressräume dieser Operationen stattfinden. Unterstützt werden nur zweiseitige asynchrone MPI-Operationen, insofern bietet *SyncChecker* lediglich eine prozess-lokale Sicht.
UPC-Spin [Ebn11] und CAF-Spin [AMS12] verifizieren PGAS-Programme statisch anhand eines gegebenen Modells. Der Quellcode wird in eine Modellbeschreibungssprache übersetzt. Das Resultat wird zur Ermittlung potentieller data races genutzt.
UPC-Thrille [PSHI11, Par12] findet data races in UPC-Programmen. Dazu verwendet das Werkzeug das Verfahren des aktiven Testens. Im Fall von *UPC-Thrille* besteht das aktive Testen aus zwei Phasen:

1. In der Voraussagephase (*Race prediction phase*) wird eine Programmausführung analysiert und eine Datenbank mit potentiellen data races angelegt.

2. In der Testphase (*Race confirmation phase*) wird das Programm ein zweites Mal kontrolliert ausgeführt. Erreicht die Ausführung eines Threads ein potentielles data race, so wird dieser Thread angehalten und anhand der Ausführung der anderen Threads festgestellt, ob dieses data race tatsächlich auftritt.

Das in der Voraussagephase verwendete Modell zur Berechnung potentieller data races unterstützt nur die von UPC bereitgestellten kollektiven Synchronisationsmechanismen. Asynchrone Speicherzugriffe durch nicht-blockierende Kommunikationsoperationen werden in dem Modell nicht unterstützt. In der Voraussagephase werden sehr viele, auch scheinbare data races erkannt. Diese scheinbaren data races werden dann in der Testphase ausgefiltert. Beim instruktionsgenauen Testen kann dabei jeder lokale Speicherzugriff zu einem Anhalten des aktuellen Threads führen. In [Par12, Tabelle 8.2, Spalte I] wird eine dadurch bis zu 6490% längere Laufzeit angegeben.

MC-Checker [CDT+14] ist ein aus drei Komponenten bestehendes Werkzeug zum Auffinden von data races in MPI-Programmen mit einseitiger Kommunikation. Die erste Komponente führt eine statische Programmanalyse durch und instrumentiert MPI-Funktionsaufrufe sowie Zugriffe auf lokale Variablen, die potentiell innerhalb eines MPI-Windows liegen. Die zweite Komponente führt das instrumentierte Programm aus und generiert ein Trace. Der durch die Instrumentierung induzierte Laufzeit-Overhead wird mit 25-147% angegeben. Die dritte Komponente wertet das Trace aus und findet auftretende data races

- innerhalb einer Epoche auf einem Knoten,
- prozessübergreifend aufgrund konkurrierender MPI-Operationen über einen Speicherbereich, und
- aufgrund eines lokalen Speicherzugriffs und einer entfernten MPI-Operation über denselben Speicherbereich.

Das der Auswertung zugrundeliegende Modell ist ein Task Graph, in dem kollektive und gerichtete Synchronisationsbeziehungen zwischen den MPI-Prozessen dargestellt werden. Die Berechnung zusammengehörender Synchronisationsoperationen geschieht allerdings lediglich anhand der aufgezeichneten Operationsreihenfolge. Ein Test auf nichtdeterministische Synchronisationsbeziehungen findet nicht statt. Asynchrone Speicherzugriffe durch nicht-blockierende Kommunikationsoperationen werden als Zugriffe parallel zum Prozess repräsentiert. Die Zugriffszeit wird in dem Fall von der Epoche begrenzt.
Mit *UPC-Thrille* und *MC-Checker* sind dem Autor nur zwei Werkzeuge bekannt, die parallele Anwendungen mit gemeinsam genutztem Adressraum unter Einbeziehung aller Speicherzugriffe analysieren. Das Anwendungsspektrum beider Werkzeuge beschränkt sich jedoch auf homogen parallelisierte Programme. Sie können außerdem nur für das Finden von data races eingesetzt werden. Als Ausgabe erhält der Programmierer die problematische Codezeile und eventuell den Grund der Wettlaufsituation. Die Synchronisationsbeziehungen zwischen Threads insbesondere im Zusammenspiel mit asynchronen PGAS-Operationen werden nicht dargestellt. Eine solche Darstellung kann aber gerade für die Untersuchung von Wettlaufsituationen sehr nützlich sein, da damit nicht nur die Existenz eines data races, sondern auch dessen Ursache dem Programmierer zugänglich gemacht wird. Ein wie in dieser Arbeit entwickeltes Werkzeug, welches für hybrid parallele Programme mit einseitiger asynchroner Kommunikation eine umfassende Korrektheitsanalyse von Synchronisationsbeziehungen und Speicherzugriffen vornehmen kann, ist dem Autor nicht bekannt.

3.2.3 Effizienzanalyse von Speicherzugriffen

Neben dem Auffinden von Programmfehlern kann Speicherzugriffsanalyse auch zur Performance-Optimierung verwendet werden. Dabei muss berücksichtigt werden, dass das Messen von Speicherzugriffen aufgrund ihres sehr häufigen Auftretens das Laufzeitverhalten des untersuchten Programms stark verzerren kann. Aus diesem Grund spielt bereits die Art der Erfassung von Speicherzugriffen für die Analysemethode eine wichtige Rolle. Bestehende Werkzeuge können in zwei Gruppen unterteilt werden, die sich in der Erfassungsart unterscheiden.

Die erste Gruppe dieser Werkzeuge instrumentiert Instruktionen des untersuchten Programms und zeichnet so die Speicherzugriffe auf. Die Messung erfolgt entweder in einer simulierten Umgebung oder die Auswertung beschränkt sich auf eine qualitative Analyse der gesammelten Daten.
MemSpy [GAM95] nutzt einen Mikroprozessor-Simulator, in dem neben Speicherzugriffs-Instruktionen auch Funktionsaufrufe und Synchronisationsoperationen behandelt werden. Die Simulation bildet die Speicherhierarchie eines Multicore-Prozessors ab und kann neben ineffizienten Cache-Zugriffen auch ineffizientes Sharing bzw. *false sharing* zwischen Threads finden. Die vollständige Instrumentierung eines Programms führte zu einer 20-100fachen Verlangsamung. Deswegen wurde das Werkzeug um Sampling erweitert, so dass nur noch Speicherzugriffe in vorgegebenen Zeitintervallen erfasst wurden. Dadurch konnte die Verlangsamung auf das 3-50fache gesenkt werden.
In [YBD01] wird der Quellcode des Programms instrumentiert, so dass nur Speicherzugriffe auf Programm-Variablen erfasst werden. Bei der Ausführung wird ein konfigurierbarer Cache simuliert. Die Ergebnisse werden in einer Bitmap visualisiert, wobei jedes Pixel einen Speicherzugriff repräsentiert. Der Verlauf der Speicherzugriffe in der Bitmap wird als eine an den Bildgrenzen umgebrochene Zeile dargestellt. Die Pixel sind farbcodiert und können unterschiedliche Aspekte des Cache-Verhaltens, wie z. B. die Art eines *cache misses* oder die *reuse distance* darstellen.
Cachegrind [Net04] simuliert eine zweistufige Cache-Hierarchie und erstellt Statistiken über die Nutzung des Instruktions- und Datencaches sowie optional über die Güte der Sprungvorhersage. Das Werkzeug ist Teil des Valgrind-Pakets [NS07], welches ein Programmiergerüst speziell für die feingranulare Instrumentierung von Anwendungen bereitstellt.
QUAD [OMGB10] nutzt das Instrumentierungs-API PIN [LCM+05], um zur Laufzeit alle Speicherzugriffe des zu untersuchenden Programms zu erfassen. Die Analyse trifft qualitative Aussagen über den Datenfluss des Programms. Dazu werden die Art der Zugriffe (lesend und/oder schreibend) und deren Adressen in Zusammenhang mit den jeweils ausgeführten Funktionen gebracht. Mit diesen Daten werden Erzeuger-Verbraucher-Beziehungen zwischen Funktionen hergestellt. Ebenfalls möglich ist die Identifizierung voneinander unabhängiger Funktionen, die potentiell parallel ausgeführt werden können.
In [SFS+11] wird eine Speicherzugriffsanalyse für MPI-Programme durchgeführt. Das Werkzeug zeichnet Speicherzugriffe und MPI-Funktionen auf. Aufeinanderfolgende Speicherzugriffe werden in Berechnungsblöcken zusammengefasst. Die anschließende Datenflussanalyse identifiziert potentiell parallel ausführbare Berechnungsblöcke und deren Abhängigkeiten zueinander.
Die zweite Gruppe von Werkzeugen zur Effizienzanalyse von Speicherzugriffen nutzt Hardware-Erweiterungen zur Aufzeichnung. Die Adressen der Zugriffe werden gesampelt, so dass nur Stichproben aufgezeichnet werden. Für diese Stichproben werden jedoch die tatsächlich aufgetretenen Performance-Eigenschaften untersucht. Da der Overhead aufgrund der Hardware-Unterstützung und des Samplings gering ist, können auch die Zeitstempel in die Analyse mit einbezogen werden.
HPCToolkit [LMC13] sampelt Speicherzugriffe in Multithreaded-Programmen und speichert diese zusammen mit dem jeweiligen Aufrufkontext. Zusätzlich werden Speicherallokation und -deallokation aufgezeichnet, so dass die Adressen der Speicherzugriffe Programmvariablen und NUMA-Domänen zugeordnet werden können. Aus den gesammelten Daten werden Statistiken zur Latenz für jeden Variablenzugriff in jeder Codezeile erstellt. Des Weiteren ist es möglich,

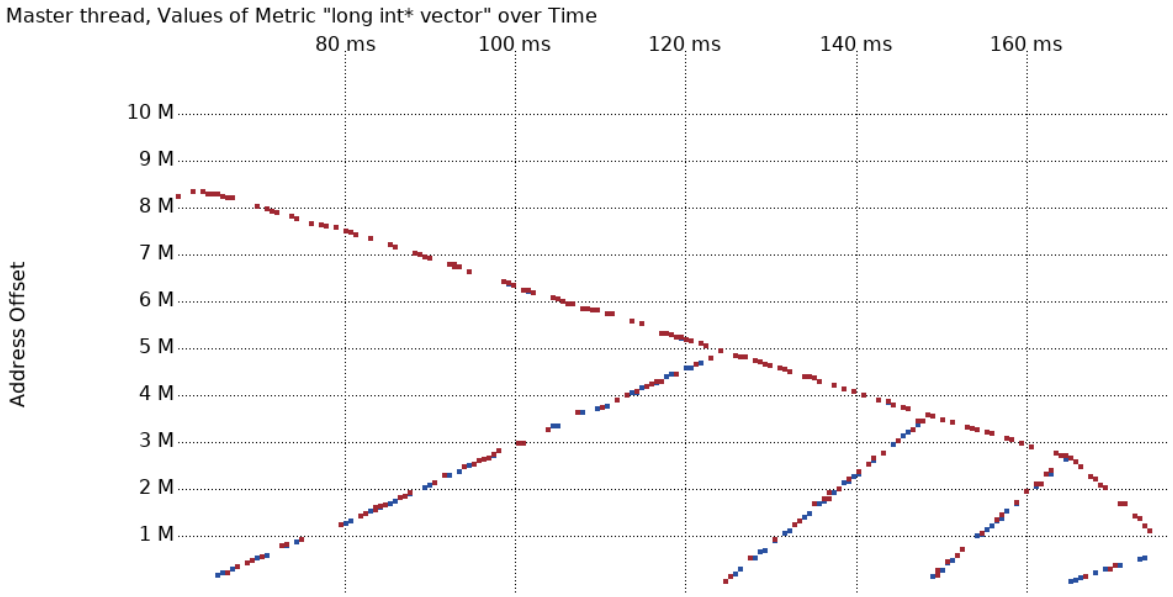


Abbildung 3.1: Speicherzugriffsmuster der ersten vier Rekursionsschritte des Quicksort-Verfahrens (entnommen aus [Wer16])

Codestellen mit häufigen Zugriffen über NUMA-Grenzen hinweg zu identifizieren.

In [Wer16] wird das Sampling von Speicherzugriffen mit Score-P [KRM+12] vorgenommen. Die Darstellung der aufgezeichneten Adressen kann in einem Zeit-/Adressdiagramm (Abbildung 3.1) vorgenommen werden. Dieses Diagramm zeigt für jeden gesampelten Zugriff einen Punkt an der entsprechenden Zeit-/Adresskoordinate. Zusätzlich können die NUMA-Charakteristiken der Speicherzugriffe ausgewertet und in einer Zeitleistendarstellung in Vampir [KBD+08] visualisiert werden.

In [SLG+15] werden zwei Aufzeichnungsverfahren miteinander kombiniert. Speicherzugriffe werden mit Hardware-Unterstützung gesamplet, wobei zu jedem Ereignis nur die Adresse, der Zeitstempel und die Latenz gespeichert werden. Weitere Performance-Metriken und der Aufrufkontext werden in einem parallelen Sampling in größeren Zeitintervallen aufgezeichnet, so dass der Overhead der Messung sehr klein wird. Die Daten werden ebenfalls in einem Zeit-/Adressdiagramm präsentiert (Abbildung 3.2). Die Farbe eines Punktes zeigt die gemessene Latenz des Zugriffs an. Die Zeitachse des Diagramms zeigt zusätzlich auch die durchlaufenen Funktionen und Performance-Statistiken wie z. B. die durchschnittliche Cache-Miss-Rate.

MemAxes [GGJ+17] stellt verschiedene Möglichkeiten der Visualisierung von gesampelten Speicherzugriffen bereit. Das Werkzeug bietet mehrere Ansichten, die für die jeweiligen Analyseziele konzipiert wurden. Die Performance des Gesamtsystems wird in einem aus konzentrischen Ringsegmenten bestehendem Kreisdiagramm dargestellt. Die äußeren Ringsegmente repräsentieren die Rechenkerne, der innere Ring den Hauptspeicher. Die dazwischen liegenden Ringsegmente repräsentieren die verschiedenen Cache-Stufen, wobei die Anordnung in Bezug auf die Rechenkerne der Hardware-Topologie entspricht. Die Farbe eines Segments codiert die Rechenlast der entsprechenden Komponente. Zu diesem Diagramm können verschiedene Histogramme und Quellcode-Ansichten geöffnet werden.

Viele Werkzeuge zur Effizienzanalyse von Speicherzugriffen konzentrieren sich auf die Untersu-

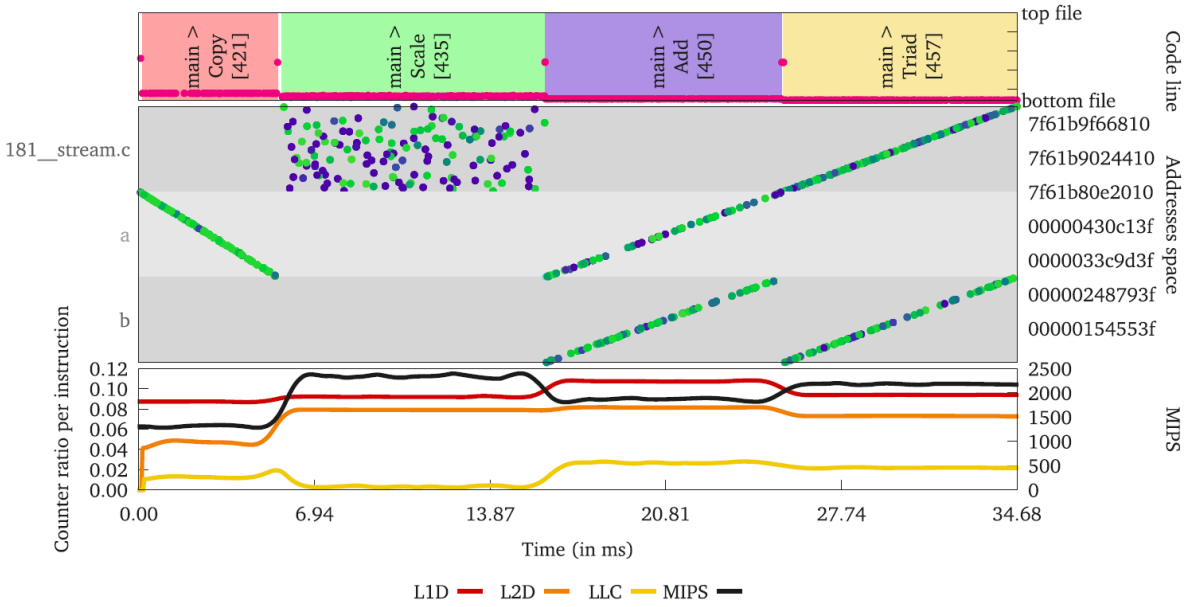


Abbildung 3.2: Zeit-/Adressdiagramm für gesampelte Speicherzugriffe und ihre Performance-Charakteristiken (entnommen aus [SLG+15])

chung des Cache-Verhaltens. Darüber hinaus finden sich einige wenige Ansätze (z. B. *QUAD*) zur Rekonstruktion des Datenflusses einer Anwendung, um daraus Optimierungsmöglichkeiten abzuleiten. Werkzeuge, welche die Effizienz von Speicherzugriffen speziell in PGAS-Systemen analysieren, sind dem Autor dagegen nicht bekannt. In solchen Systemen ist ein optimales Zusammenspiel von asynchronen und synchronen Speicherzugriffen zum Erreichen einer guten Überlagerung von Berechnung und Kommunikation ein in der bisherigen Werkzeugentwicklung noch nicht betrachteter Analyseaspekt.

3.3 Analysewerkzeuge für PGAS-Programme

Neben den in Abschnitt 3.2.2 genannten Arbeiten zur Speicherzugriffsanalyse von PGAS-Programmen existieren nur wenige Debugger, die spezielle Mittel für PGAS-Programme bereitstellen. *TotalView*® [tot18] ist ein Debugger, der für UPC-Programme die Verteilung von Feld-Variablen auf PGAS-Prozesse darstellen kann. In [DAC+14] wird ein Debugger beschrieben, der vergleichende Tests für nach UPC portierte Programme vornehmen kann. Darüber hinaus existierende Debugger für PGAS-Programme bieten vor allem konventionelle Analyse-Techniken wie z. B. Variableninspektionen an Haltepunkten.

Für die Effizienzanalyse von PGAS-Programmen existiert eine Reihe von Werkzeugen, mit denen die Kosten von PGAS-Operationen untersucht werden können. In [TK12] wird eine Erweiterung von HPCToolkit vorgestellt, mit der eine daten-zentrierte Performance-Analyse vorgenommen werden kann. In der Aufzeichnungsphase werden Zugriffe auf entfernte Variablen gesamplet. Aus diesen Daten werden ein Zeitprofil der aufgerufenen Funktionen und ein Profil der von diesen Funktionen übertragenen Datenmengen erstellt. In einer Zeitleistenvisualisierung der Prozesse sind entfernte Zugriffe auf Variablen im gemeinsamen Adressraum farblich kodiert, so dass für jede Variable die Menge dieser Zugriffe und deren jeweilige Dauer ersichtlich sind.

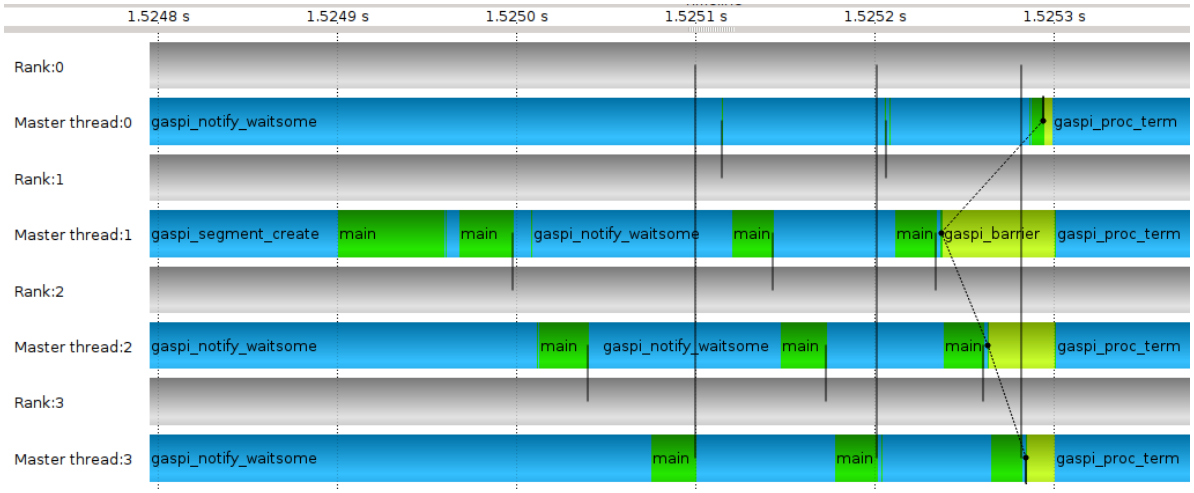


Abbildung 3.3: Darstellung von GASPI-Operationen und Prozessbeziehungen in Vampir (entnommen aus [Her15])

Die hardware-gestützte Messung des Energieverbrauchs von PGAS-Datenübertragungen wird in [JSC14] verwendet, um den Einfluss unterschiedlicher Kommunikationsstrategien auf den Energieverbrauch eines PGAS-Programms zu untersuchen. Die dort vorgenommene Analyse liefert zu einem Programmlauf den auf Prozessor und Speicher aufgeschlüsselten Energiebedarf.
Vampir [KBD+08] stellt die Dauer von PGAS-Operationen in einer Zeitleistenvisualisierung dar. Um den speziellen Anforderungen von hybriden PGAS-Programmen gerecht zu werden, wird für jeden Rank ein zusätzlicher Balken dargestellt, der den Speicher des Prozesses symbolisiert. In Abbildung 3.3 sind diese Balken grau dargestellt. Sie werden als Ziel von Kommunikationsoperationen genutzt, da in einem PGAS-Programm für den einseitigen Transfer von Daten kein zweiter direkt beteiligter Thread angegeben werden kann.
Parallel Performance Wizard (PPW) [SBG08] ist ein vielseitig einsetzbares Performance-Analyse-Werkzeug für PGAS-Programme. PPW arbeitet unabhängig von der PGAS-Implementierung, da aufgezeichnete Ereignisse zu allgemeinen PGAS-Operationen abstrahiert werden. Es existieren Backends für UPC und OpenSHMEM. Eingebaut in PPW sind mehrere Möglichkeiten der statistischen Auswertung der Aufzeichnung. Darüber hinaus können die verallgemeinerten Aufzeichnungsdaten konvertiert werden, so dass auch eine Zeitleistenvisualisierung, z. B. mit Vampir möglich ist.
Alle vorgestellten Werkzeuge verzichten auf eine Einbeziehung lokaler Speicherzugriffe in die Analyse. Dem Autor ist kein Ansatz bekannt, der das Zusammenspiel von lokalen und entfernten Speicherzugriffen in einem PGAS-System unter Performance-Gesichtspunkten betrachtet.

4 Modellierung von asynchronen einseitigen Kommunikationssystemen

Everything should be made as simple as possible, but not simpler. [ECD10]

Einseitige Kommunikationssysteme sind in mehreren Varianten verfügbar. Sie existieren als PGAS-Sprachen (z.B. UPC), als PGAS-APIs (z.B. GASPI, OpenSHMEM) sowie als Erweiterungen bestehender APIs (z.B. in MPI-3). Ein Modell für einseitige Kommunikationssysteme muss sowohl die essentiellen Merkmale einseitiger Kommunikation abbilden als auch variantenspezifische Eigenschaften abstrahieren. Auf diese Weise kann das Modell zur Entwicklung von allgemein nutzbaren Werkzeugen verwendet werden, die nicht an ein bestimmtes Kommunikationssystem gebunden sind.
Das in dieser Arbeit entwickelte Modell repräsentiert die Ausführung eines hybrid parallelen Programms. Das Modell berücksichtigt die zwei Parallelitätsebenen Thread und Prozess. Die Ausführung einer einzelnen Programmanweisung ist immer genau einem Thread zugeordnet. Ein Prozess enthält eine Anzahl von Threads, wobei jeder Thread gleichberechtigt Kommunikations- und Synchronisationsfunktionen auf Prozessebene aufrufen kann. Jedem Prozess ist weiterhin eineindeutig ein Rank zugeordnet. Ein hybrid paralleles Programm im Sinne dieser Arbeit besteht aus einer Menge solcher Prozesse.
Im Modell werden alle wesentlichen Ereignisse einer Programmausführung dargestellt. Dazu gehören die von den Threads ausgeführten *direkten* Speicherzugriffe in den lokalen Speicher und in PGAS-Segmente. Weiterhin gehören dazu Funktionsaufrufe und die asynchronen Ereignisse, die durch das verwendete Kommunikationssystem ausgelöst werden. Ebenfalls dargestellt werden thread- und prozessübergreifende kausale Abhängigkeiten zwischen einzelnen Programmereignissen.

4.1 Synchronisationsbeziehungen in hybrid parallelen Programmausführungen

Prozesse und Threads koordinieren ihre zeitlichen Abläufe untereinander mit Hilfe von Synchronisationsnachrichten. Synchronisation kann gerichtet oder ungerichtet erfolgen. Eine gerichtete Synchronisation liegt vor, wenn Sender und Empfänger einer Nachricht a priori, d.h. beim Aufruf der entsprechenden Synchronisationsoperationen bekannt sind. Eine solche Synchronisation bezeichnet eine kausale Abhängigkeit im Programmablauf und induziert eine happened-before-Relation vom Sender zum Empfänger. Beispiele für gerichtete Synchronisationen sind point-to-point-Synchronisation und kollektive Synchronisation.
Ungerichtete Synchronisation findet beim wechselseitigen Ausschluss mit Hilfe von kritischen Programmabschnitten statt. In einer gegebenen Programmausführung existiert eine happened-

before-Relation vom Verlassen eines kritischen Abschnitts zum nächsten Eintritt in diesen Abschnitt. Diese happened-before-Relation ist jedoch nicht stabil in Bezug auf zeitliche Variationen in anderen Ausführungen desselben Programms. Zeitliche Verschiebungen können zu einer veränderten Reihenfolge der Abarbeitung der kritischen Abschnitte führen. Der entstehende Nichtdeterminismus ist im Allgemeinen intendiert (siehe auch Abschnitt 2.3).
Die Aufzeichnung einer Programmausführung stellt im Allgemeinen nur die Abfolge der Programmereignisse innerhalb der Threads bereit. Die Berechnung der happened-before-Relationen zwischen Programmereignissen unterschiedlicher Threads stellt deswegen eine zentrale Aufgabe bei der Erstellung eines Task Graphen dar. Dabei sind verschiedene Herausforderungen zu bewältigen:

1. Aus der Aufzeichnung eines einzelnen Synchronisationsereignisses kann nicht notwendigerweise direkt auf alle an der Synchronisation beteiligten Ereignisse geschlossen werden.

2. Die berechneten happened-before-Relationen sollen die logischen Abhängigkeiten innerhalb des Programms abbilden. Existieren Sychronisations-Races, so sollten diese erkannt und lokalisiert werden.

3. In hybrid parallelen Programmen teilen sich die Threads eines Prozesses bestimmte Synchronisationsressourcen.

Die erste Herausforderung leitet sich aus den zuverlässig verfügbaren Informationen ab, die eine Programmaufzeichnung überhaupt liefern kann. Vorausgesetzt wird, dass lokal verfügbare Informationen, wie der Name einer Funktion, die Werte der Argumente und der Rückgabewert aufgezeichnet werden können. Die Ermittlung darüber hinausgehender Informationen ist jedoch systemabhängig. In GASPI ist es z.B. standardmäßig nicht möglich, anhand einer empfangenen Notifikation den Rank des Senders zu ermitteln. Dazu müsste entweder eine spezielle Debug-Implementierung mit Piggybacking eingesetzt werden, oder die Beziehungen zwischen Synchronisationsereignissen werden anhand der Ausführungszeiten der Programmereignisse berechnet. Allerdings sind solche Methoden nur geeignet, um die Synchronisationsbeziehungen des gegebenen Programmlaufs im Rahmen der konkreten Ausführungszeiten der Ereignisse zu ermitteln. Nur mit diesen Verfahren kann also die zweite Herausforderung nicht bewältigt werden. Dafür muss zusätzlich getestet werden, ob die aufgezeichneten Synchronisationsbeziehungen deterministisch sind oder ob Sychronisations-Races existieren.
Die dritte Herausforderung soll anhand eines Beispiels erklärt werden. In Abbildung 4.1a sind zwei Prozesse dargestellt, die jeder eine Barriere zweimal aufrufen. Im Task Graph sind die beiden Ereignisse *Betreten* und *Verlassen* der Barriere dargestellt. Die happened-before-Relationen zwischen den Prozessen sind mit gewellten Linien dargestellt. Aus einer Programmaufzeichnung können diese Beziehungen berechnet werden, indem für jeden Prozess die Barrieren-Ereignisse anhand ihrer Aufrufreihenfolge durchnummeriert werden. Ereignisse mit gleicher Nummer gehören dann zusammen und werden entsprechend mit happened-before-Relationen verbunden. Dieses Verfahren ist jedoch für hybrid parallele Programmausführungen nicht mehr möglich. In Abbildung 4.1b besteht Prozess 1 aus 2 Threads, von denen jeder die Barriere genau einmal aufruft. Nachdem Thread 2 die Barriere verlassen hat, sendet er eine Nachricht an Thread 1.

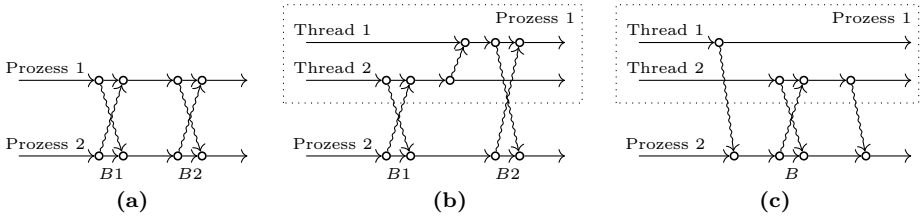


Abbildung 4.1: Interaktion von kollektiver und point-to-point Synchronisation in hybrid parallelen Programmausführungen

Thread 1 betritt die Barriere erst nach Empfang dieser Nachricht. Um die logisch notwendige Aufrufreihenfolge der Barrieren in Prozess 1 festzustellen, ist also die gerichtete Synchronisation von Thread 2 zu Thread 1 zu berücksichtigen. Abbildung 4.1c illustriert den umgekehrten Fall. Thread 1 und Thread 2 senden jeweils dieselbe Nachricht an Prozess 2. Aufgrund der zwischengeschalteten Barriere wird Prozess 2 immer zuerst die Nachricht von Thread 1 empfangen. Um die Reihenfolge und Eindeutigkeit der point-to-point-Synchronisationen festzustellen, sind in diesem Fall also auch die kollektiven Synchronisationsereignisse zu berücksichtigen. Aus diesen Betrachtungen folgt, dass eine Berechnung von happened-before-Relationen für kollektive und point-to-point-Synchronisationen nicht unabhängig voneinander erfolgen kann.

Das im Folgenden entwickelte Verfahren berechnet gerichtete Synchronisationsbeziehungen in hybrid parallelen Programmen. Es wurde bereits in [KMPN16] veröffentlicht und wird im Folgenden ausführlicher und präzisiert beschrieben. Das zugrundeliegende Modell unterstützt als Synchronisations-Primitive blockierende und nicht-blockierende kollektive sowie point-to-point-Synchronisation. Es ist insbesondere für die Analyse von GASPI-Programmen geeignet, da alle GASPI-Synchronisationsfunktionen direkt in die unterstützten Primitive umgesetzt werden können. Des Weiteren kann das Verfahren Nichtdeterminismen erkennen, die sich aus fehlerhaften Synchronisationsabfolgen ergeben.

Die entwickelte Theorie basiert auf Überlegungen aus [NBDK96]. Insbesondere wird Satz 2 dort bereits für homogen parallele MPI-Programme, die aus `MPI_Send` und `MPI_Recv`-Aufrufen bestehen, bewiesen. Das hier eingeführte Modell generalisiert die Synchronisationsoperationen und erweitert diese auf kollektive Synchronisationen. Außerdem werden die Besonderheiten hybrid paralleler Programmausführungen in die Theorie mit einbezogen.

4.1.1 Synchronisationsmodell

Im point-to-point-Synchronisationsmodell bildet das *Flag* das grundlegende Konzept zur Kommunikation zwischen Threads. Die drei klassischen Basisoperationen *POST*, *WAIT* und *CLEAR* über ein Flag wurden in Abschnitt 3.1 (Seite 23) erläutert. Abhängig vom verwendeten Programmiermodell kann auf Flags thread- oder auch prozessübergreifend zugegriffen werden.

Kollektive Synchronisationen können innerhalb des Flag-Modells durch eine Erweiterung der *WAIT*-Operation beschrieben werden. Jeder an einer Kollektive beteiligte Thread bzw. Prozess besitzt ein eigenes Flag. Zum Beispiel besitzt jeder an einer OpenMP-Kollektive beteiligte Thread ein thread-lokales Flag für diese Kollektive. Das einer GASPI-Kollektive zugehörige Flag eines Prozesses wird von allen Threads des Prozesses gemeinsam benutzt. Die *WAIT*-Operation einer Kollektive unterbricht die Ausführung des Threads solange, bis der Zustand

aller an der Kollektive beteiligten Flags *gesetzt* ist. Es handelt sich also um eine konjunktive *WAIT*-Operation [Coo03].
Wird eine blockierende Kollektive von einem Thread betreten, so wird zuerst eine *POST*-Operation auf das eigene Flag durchgeführt. Danach wird die konjunktive *WAIT*-Operation durchgeführt und es wird gewartet, bis alle teilnehmenden Threads bzw. Prozesse die Kollektive betreten haben. Vor dem Verlassen der Kollektive wird dann noch eine *CLEAR*-Operation auf das eigene Flag ausgeführt. In einer blockierenden Kollektive sind die drei Basisoperationen also Bestandteile einer zusammengesetzten Operation. In einer nicht-blockierenden Kollektive wird die *POST*-Operation in eine dedizierte Funktion ausgelagert. Die *WAIT*- und *CLEAR*-Operationen werden in einer Kollektive immer verbunden zu einer atomaren Operation ausgeführt.
Ein wichtiger Schritt hin zu einem berechenbaren Synchronisationsmodell besteht nun darin, auch für die gerichtete Synchronisation eine Verknüpfung der *WAIT* und *CLEAR*-Operationen zu einer einzigen Operation *WAITCLEAR* vorzunehmen. Damit besteht das im Weiteren verwendete Synchronisationsmodell aus zwei prinzipiellen Operationen:

- *POST(f)* oder P: entspricht der *POST*-Operation des klassischen Modells und setzt den Zustand des Flags f auf *gesetzt*.
- *WAITCLEAR(f)* oder W: unterbricht die weitere Ausführung des Threads solange, bis der Zustand des Flags f *gesetzt* ist. Wenn f zu einer Kollektive gehört, dann wird die weitere Ausführung solange unterbrochen, bis die jeweiligen Flags aller an der Kollektive beteiligten Threads oder Prozesse den Zustand *gesetzt* haben. Vor dem Verlassen von W wird f auf *gelöscht* geändert.

In diesem Modell kann gerichtete Synchronisation auch als ein Spezialfall der kollektiven Synchronisation begriffen werden – als nicht-blockierende Kollektive, in der die *WAITCLEAR*-Operation genau einen Teilnehmer hat. Die *WAITCLEAR*-Operation muss die in ihr enthaltenen *WAIT*- und *CLEAR*-Operationen nicht atomar ausführen. Für die folgenden Betrachtungen ist es lediglich entscheidend, dass sich zwischen diesen beiden Basisoperationen keine weitere Synchronisationsoperation befindet.
Wenn eine *WAITCLEAR*-Operation W aufgrund einer oder – im Falle einer Kollektive – mehrerer *POST*-Operationen P die Thread-Ausführung fortsetzt, so wurde W von den P *getriggert* und es gilt $P \prec W$. Wird eine *WAITCLEAR*-Operation getriggert, so wird der Thread weiter ausgeführt werden. Trigger gehen also nicht verloren. Eine *WAITCLEAR*-Operation kann jedoch wie auch eine *WAIT*-Operation nicht in die Vergangenheit schauen, sondern betrachtet den Zustand des Flags erst ab dem Startzeitpunkt der Operationsausführung.
Die Berechnung der Synchronisationsbeziehungen anhand einer Programmaufzeichnung erfolgt nun, indem zuerst die Ereignisse auf die *POST*- und *WAITCLEAR*-Operation reduziert werden und dann anhand der Abfolge dieser Operationen die konkreten happened-before-Relationen berechnet werden. Gegeben sei also ein partieller Task Graph in Form einer Programmaufzeichnung $\mathcal{P}^T = \left\langle E, \prec^T \right\rangle$. Jeder Task in E ist entweder eine *POST(f)* oder eine *WAITCLEAR(f)*-Operation. Das Flag f ist Teil der Eingabe und enthält auch die Informationen über den genauen Typ der Synchronisation. Zusätzlich enthalten die *WAITCLEAR*-Operationen die Information,

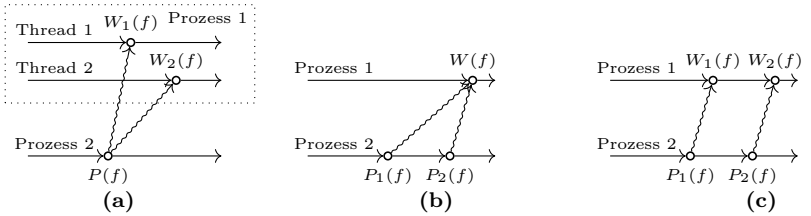


Abbildung 4.2: Verschiedene Arten von Synchronisations-Races

ob sie während des Programmlaufs getriggert wurden. Eine nicht getriggerte *WAITCLEAR*-Operation kann am Ende eines Threads auftreten. Dieses Merkmal weist auf einen abnormalen Programmabbruch hin, der beispielsweise in Folge von Deadlocks vollzogen werden musste. Die Relation $\prec^T$ bezeichnet die sequentielle Ausführungsreihenfolge der Tasks innerhalb eines Threads. Diese Relation ist direkt aus der Programmaufzeichnung ableitbar.
Um die Synchronisationsbeziehungen der Threads untereinander abzubilden, muss aus $\mathcal{P}^T$ eine Programmausführung $\mathcal{P} = \left\langle E, \prec^T \cup \prec^S \right\rangle$ konstruiert werden. In $\mathcal{P}$ bezeichnet die Relation $\prec^S$ einen Trigger einer *WAITCLEAR*-Operation durch eine *POST*-Operation. Kann nur genau eine Programmausführung $\mathcal{P}$ konstruiert werden, dann ist der durch $\mathcal{P}^T$ repräsentierte Programmablauf deterministisch. Ist mindestens eine zweite Programmausführung $\dot{\mathcal{P}} = \left\langle E, \prec^T \cup \dot{\prec}^S \right\rangle$ konstruierbar, dann ist der durch $\mathcal{P}^T$ repräsentierte Programmablauf nicht-deterministisch und enthält demzufolge mindestens ein Synchronisations-Race.

4.1.2 Synchronisations-Races

Parallele Programme können verschiedene Formen nicht-deterministischen Verhaltens aufweisen. Wichtige Ausprägungen wurden bereits in Abschnitt 2.3 erläutert. Synchronisations-Races bilden in dieser Aufzählung eine eigene Kategorie. Solche Races führen zu nichtdeterministischen happened-before-Relationen zwischen gerichteten Synchronisationsoperationen. In dieser Arbeit werden diese Races immer als Programmierfehler betrachtet.
Im hier verwendeten Synchronisationsmodell kann ein Synchronisations-Race nur entstehen, wenn mehrere Synchronisationsoperationen so auf ein Flag zugreifen, dass die sich ergebenden Synchronisationsbeziehungen nicht mehr eindeutig sind. In Abbildung 4.2a führt Prozess 2 eine *POST*-Operation aus. Es ist jedoch nicht bestimmbar, ob Thread 1, Thread 2 oder sogar beide Threads diese *POST*-Operation feststellen werden. Das hängt von den tatsächlichen Zeitpunkten ab, an denen P, W_1 und W_2 ausgeführt werden. Wird zuerst P und dann W_1 (bzw. W_2) komplett ausgeführt, so sorgt das in *WAITCLEAR* eingebaute *CLEAR* dafür, dass W_2 (bzw. W_1) die *POST*-Operation nicht feststellen kann und somit auf ein späteres *POST* warten wird. Werden W_1 und W_2 genau gleichzeitig ausgeführt, so ist es möglich, dass beide Threads ihre Ausführung aufgrund ein und derselben *POST*-Operation fortsetzen. Abbildung 4.2b illustriert ein Synchronisations-Race zweier *POST*-Operationen auf eine *WAITCLEAR*-Operation. Wenn Prozess 1 W bereits vor der Ausführung von P_1 betreten hat, dann wird Prozess 1 durch P_1 mit der Ausführung fortfahren. Nach der Ausführung von P_2 ist $f = 1$. Wenn jedoch Prozess 1 W erst ausführt, nachdem Prozess 2 bereits P_2 ausgeführt hat, so gilt schließlich $f = 0$. Abbildung 4.2c erweitert das Beispiel aus Abbildung 4.2b. Auf den ersten Blick scheint die Ausführungsreihenfolge wohl-definiert zu sein, da $P_1 \prec W_1$ und $P_2 \prec W_2$ gilt. Wenn jedoch P_1 und P_2 bereits vor W_1

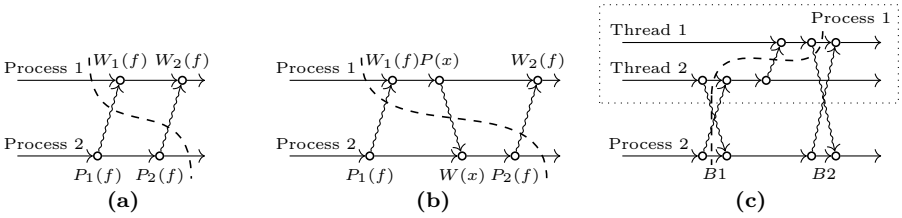


Abbildung 4.3: Grenzlinien konsistenter (a) und inkonsistenter (b,c) globaler Programmzustände

ausgeführt wurden, dann wird die Ausführung von W_2 blockiert, da f bereits durch W_1 wieder gelöscht wurde.

Formal definiert diese Arbeit ein Synchronisations-Race als einen speziellen globalen Programmzustand. Ein globaler Programmzustand auf $\mathcal{P}$ kann als eine Grenzlinie gesehen werden, die zwischen den Tasks aller Threads gezogen wird. Alle Tasks vor der Grenzlinie wurden bereits ausgeführt. Tasks direkt hinter der Grenzlinie werden vom jeweiligen Thread als Nächstes ausgeführt. Solche Tasks werden im Folgenden als *aktiv* bezeichnet. Ein konsistenter globaler Programmzustand ist eine Grenzlinie, die alle Threads gleichzeitig erreichen können [CL85]. Eine solche Grenzlinie kann also nur von happened-before-Pfeilen in Richtung der Programmausführung überquert werden. Außerdem müssen alle *WAITCLEAR*-Operationen vor einer konsistenten Grenzline getriggert sein.

Definition 11. *Eine Programmausführung $\mathcal{P}$ enthält ein* Synchronisations-Race *genau dann, wenn ein konsistenter globaler Programmzustand existiert, so dass eine aktive WAITCLEAR-Operation auf einem Flag f existiert und*

1. *eine weitere WAITCLEAR-Operation auf f aktiv ist und eine POST-Operation auf f vor der Grenzlinie existiert, die keine WAITCLEAR-Operation vor der Grenzlinie triggert; oder*

2. *zwei POST-Operationen auf f vor der Grenzlinie existieren, die keine WAITCLEAR-Operation vor der Grenzlinie triggern.*

Der erste Punkt von Definition 11 besagt, dass *WAITCLEAR*-Operationen auf dasselbe Flag nicht parallel ausgeführt werden dürfen. Für je zwei *WAITCLEAR*-Operationen W_1 und W_2 auf ein Flag muss also $W_1 \prec W_2 \vee W_2 \prec W_1$ gelten. Entsprechend besagt der zweite Punkt der Definition, dass auch parallele *POST*-Operationen ein Synchronisations-Race darstellen. Darüber hinaus impliziert dieser Punkt eine weitere Eigenschaft des Task Graphen, die im Folgenden hergeleitet wird.

Abbildung 4.3a greift Abbildung 4.2c wieder auf. Die gestrichelte Linie stellt die Grenzlinie dar. Diese gehört zu einem konsistenten globalen Programmzustand, da sie nur von Pfeilen in Richtung der Programmausführung überquert wird. Laut Definition 11 handelt es sich beim dargestellten Task Graphen um ein Synchronisations-Race: W_1 ist aktiv, P_1 und P_2 befinden sich vor der Grenzlinie und sind nicht durch eine happened-before-Relation mit einer *WAITCLEAR*-Operation vor der Grenzlinie verbunden. Im Gegensatz dazu stellt die Grenzlinie in Abbildung 4.3b keinen konsistenten globalen Programmzustand dar, da der durch die happened-before-Relation von $P(x)$ nach $W(x)$ induzierte Pfeil diese Linie entgegen der Ausführungsrichtung überquert. Abbildung 4.3c wendet das Konzept auf eine kollektive Synchronisation in einer

hybrid parallelen Ausführung an. Die eingezeichnete Grenzlinie separiert die Ereignisse *Eintritt* und *Verlassen* der beiden Barrieren-Aufrufe B_1 und B_2. Dadurch sind sowohl W_{B1} als auch W_{B2} aktiv. Allerdings ist die Grenzlinie inkonsistent, da sie aufgrund der gerichteten Synchronisation von Thread 2 zu Thread 1 von einem Pfeil entgegen der Ausführungsrichtung überquert wird. Weder in Abbildung 4.3b noch in Abbildung 4.3c ist die Konstruktion einer konsistenten Grenzlinie möglich, die alle Forderungen von Definition 11 erfüllt. Die Beispiele zeigen, dass zwei Operationen $P(x)$ und $W(x)$ nicht Teil eines Synchronisations-Races sein können, wenn $W(x) \prec P(x)$ gilt.

Satz 1. *Gegeben sei ein Task Graph, in dem keine Synchronisations-Races gemäß Definition 11.1 vorhanden sind. Sei P eine POST-Operation in diesem Graphen auf ein Flag f, W eine WAITCLEAR-Operation auf f und P und W können gleichzeitig aktiv sein. Sei außerdem P_r eine weitere POST-Operation auf f und $P_r \not\prec P$. Ein Synchronisations-Race existiert genau dann, wenn $W \not\prec P_r$.*

Beweis. Gemäß Definition 11.2 soll eine konsistente Grenzlinie konstruiert werden, so dass eine *WAITCLEAR*-Operation auf f aktiv ist und sich sowohl P als auch P_r vor der Grenzlinie befinden. Wenn P und W gleichzeitig aktiv sind, hat P keine *WAITCLEAR*-Operation vor der Grenzlinie getriggert, da Fälle nach Definition 11.1 ausgeschlossen sind. Außerdem gilt damit auch $W \not\prec P$.
$\Rightarrow$: Die Grenzlinie wird so konstruiert, dass W aktiv ist. W befindet sich also hinter der Grenzlinie. Wenn $W \prec P_r$ gilt, dann muss sich auch P_r hinter jeder konsistenten Grenzlinie befinden. Die Bedingungen von Definition 11.2 können also nicht erfüllt werden.
$\Leftarrow$: Sei $Next(P_r)$ der direkt auf P_r folgende Task. Die Grenzlinie wird so konstruiert, dass sie zwischen P_r und $Next(P_r)$ verläuft. Weiterhin wird die Grenzlinie so konstruiert, dass W aktiv ist. Dieser Schritt erfordert keine Verschiebung der bereits platzierten Grenzlinie, da $W \not\prec P_r$ laut Voraussetzung. Sollte durch diese Konstruktion P bereits vor der Grenzlinie liegen (z.B. durch $P \prec P_r$), dann sind die Bedingungen von Definition 11.2 erfüllt: W ist aktiv, P und P_r liegen vor der Grenzlinie und triggern keine *WAITCLEAR*-Operation vor der Grenzlinie. Ansonsten wird die Grenzlinie so weiter konstruiert, dass sie zwischen P und $Next(P)$ verläuft. Auch dieser Schritt erfordert keine Verschiebung der bereits platzierten Grenzlinie zur Wahrung der Konsistenz, da $W \not\prec P$ aufgrund der Anfangsüberlegungen und $P_r \not\prec P$ laut Voraussetzung. Damit sind die Bedingungen von Definition 11.2 wieder erfüllt: W ist aktiv, P und P_r liegen vor der Grenzlinie und P_r triggert keine *WAITCLEAR*-Operation vor der Grenzlinie. □

Aus Definition 11.1 und Satz 1 folgt, dass in einem race-freien Task Graphen die Abfolge der *WAITCLEAR*-Operationen auf ein bestimmtes Flag total geordnet sein muss. Weiterhin kann gezeigt werden, dass Definition 11 ausreichend ist, um Nichtdeterminismus in einem Task Graphen zu ermitteln.

Satz 2. *Wenn eine Programmausführung $\mathcal{P}$ keine Synchronisations-Races enthält, so ist $\mathcal{P}$ deterministisch.*

Beweis. Angenommen, eine Programmausführung $\mathcal{P} = \langle E, \prec \rangle$ ist frei von Synchronisations-Races. Wenn $\mathcal{P}$ nicht deterministisch ist, dann muss eine weitere Programmausführung $\dot{\mathcal{P}} =$

$\left\langle \dot{E}, \dot{\prec} \right\rangle$ existieren, die bis zu einem bestimmten Punkt dieselben Tasks und happened-before-Relationen wie $\mathcal{P}$ aufweist und sich danach unterscheidet. Sei W die erste *WAITCLEAR*-Operation auf ein Flag f, deren Trigger sich in $\mathcal{P}$ und $\dot{\mathcal{P}}$ unterscheidet. Zwei Fälle sind zu unterscheiden:

1. Seien P_1 und P_2 zwei unterschiedliche *POST*-Operationen, die W in $\mathcal{P}$ bzw. $\dot{\mathcal{P}}$ triggern. Dann gilt $W \not\prec P_1$, denn P_1 triggert W in $\mathcal{P}$. Außerdem gilt $W \not\prec P_2$, denn P_2 triggert W in $\dot{\mathcal{P}}$ und alle Tasks und Relationen vor W sind in $\mathcal{P}$ und $\dot{\mathcal{P}}$ gleich. O.B.d.A. wird $P_1 \not\prec P_2$ angenommen, denn $P_1 \prec P_2 \wedge P_2 \prec P_1$ kann nicht gelten. Damit sind die Bedingungen von Satz 1 mit $P = P_2$ und $P_r = P_1$ erfüllt. Das jedoch widerspricht der Annahme, dass $\mathcal{P}$ frei von Synchronisations-Races ist.

2. O.B.d.A. wird angenommen, dass W in $\mathcal{P}$ nicht getriggert wird, wohl aber in $\dot{\mathcal{P}}$ von einer *POST*-Operation P. Dann existiert ein W_x in $\mathcal{P}$, welches von P getriggert wurde und f gelöscht hat, bevor W ausgeführt wurde. Es gilt also $W \not\prec W_x$ in $\mathcal{P}$, denn W_x wurde getriggert, nicht aber W. Des Weiteren gilt auch $W_x \not\prec W$ in $\mathcal{P}$.

   Unterbeweis. Wenn $W_x \prec W$ in $\mathcal{P}$ gilt, dann ist W_x Teil der Programmausführung, die in $\mathcal{P}$ und $\dot{\mathcal{P}}$ gleich ist. Dann gilt $W_x \prec W$ auch in $\dot{\mathcal{P}}$, da sich erst der Trigger von W in $\mathcal{P}$ und $\dot{\mathcal{P}}$ unterscheidet. Außerdem wird W_x auch in $\dot{\mathcal{P}}$ von P getriggert. In $\dot{\mathcal{P}}$ gilt also $P \prec W_x \prec W$. Dann aber kann P in $\dot{\mathcal{P}}$ W nicht triggern, da W_x f vorher löscht. Also $W_x \not\prec W$ in $\mathcal{P}$. □

   Da also $W \not\prec W_x \wedge W_x \not\prec W$ in $\mathcal{P}$, kann eine konsistente Grenzlinie konstruiert werden, so dass W und W_x beide aktiv sind und P vor der Grenzlinie liegt. Damit sind die Bedingungen von Definition 11.1 erfüllt, was wiederum der Annahme widerspricht, dass $\mathcal{P}$ frei von Synchronisations-Races ist.

□

Satz 2 impliziert, dass für eine Programmaufzeichnung $\mathcal{P}^T$ genau eine Programmausführung $\mathcal{P}$ existiert, wenn $\mathcal{P}^T$ keine Synchronisations-Races enthält. Satz 2 impliziert auch, dass keine von Synchronisations-Races freie Programmausführung $\mathcal{P}$ existieren kann, wenn $\mathcal{P}^T$ Synchronisations-Races enthält.

4.1.3 Konstruktion des Task Graphen

Der Algorithmus zur Konstruktion von $\mathcal{P}$ aus $\mathcal{P}^T$ basiert auf der Wiederausführung der Synchronisationsereignisse. Bei diesem sogenannten Replay-Verfahren werden alle aufgezeichneten Ereignisse modellhaft erneut ausgeführt, wobei die Ausführungsreihenfolge die Semantik der Synchronisations-Operationen beachtet. Wenn während dieser modellhaften Ausführung eine *WAITCLEAR*-Operation getriggert wird, so wird eine $\prec$-Relation von der triggernden *POST*-Operation eingefügt. Eine weitere Restriktion der Ausführungsreihenfolge ist aufgrund von Satz 2 nicht notwendig. Wenn die aufgezeichnete Programmausführung frei von Synchronisations-Races ist, so wird jede mögliche Ausführungsreihenfolge zum selben Ergebnis führen. Während der Konstruktion wird der Task Graph auf auftretende Synchronisations-Races anhand der Eigenschaften in Definition 11.1 und Satz 1 getestet.

```
1  function replay_tasks (Task T) {
2    while T != nil {
3      let e = event of T
4      let r = index of T in e
5      let PWP = map[e]
6      switch type (T) {
7        case Post:
8          if PWP[r].Post != nil
9            abort and report post/post race
10         if PWP[r].PrevWait != nil and not PWP[r].PrevWait ≺ T
11           abort and report wait/post race
12         PWP[r].Post = T
13       case Wait:
14         if PWP[r].Wait != nil or
15            PWP[r].PrevWait != nil and not PWP[r].PrevWait ≺ T
16           abort and report wait/wait race
17         PWP[r].Wait = T
18     }
19     if (∀ x: PWP[x].Post != nil and PWP[x].Wait != nil) {
20       ∀ x: ∀ y:
21         add PWP[x].Post ≺ PWP[y].Wait
22       ∀ x:
23         PWP[x].PrevWait = PWP[x].Wait
24         PWP[x].Post = PWP[x].Wait = nil
25       ∀ x:
26         replay_tasks(next_Task(PWP[x].PrevWait))
27     }
28     if (type (T) == Wait) return
29     T = next_Task(T)
30   }
31 }
```

Listing 4: Das Replay-Verfahren zur Konstruktion eines Task Graphen

Listing 4 zeigt eine gekürzte Fassung des Replay-Verfahrens. Im Listing wird die einheitliche Behandlung von point-to-point und blockierender kollektiver Synchronisation gezeigt. Die Funktion `replay_tasks` führt nacheinander die Tasks eines Threads aus, bis entweder das Thread-Ende oder eine ungetriggerte *WAITCLEAR*-Operation erreicht sind. Abhängig vom Typ des gerade ausgeführten Tasks bezeichnet das Event `e` (Zeile 3) entweder das Flag (point-to-point Synchronisation) oder die Prozess- bzw. Thread-Gruppe (kollektive Synchronisation). Der Index `r` (Zeile 4) bezeichnet die Position des Threads bzw. Prozesses in der Gruppe. Für point-to-point Synchronisation gilt immer `r = 0`. Jedem Element eines Events ist eine Datenstruktur `PWP` zugeordnet, die 3 Elemente speichert. In `PWP.Wait` wird die gerade aktive *WAITCLEAR*-Operation gespeichert. `PWP.PrevWait` enthält die zuletzt getriggerte *WAITCLEAR*-Operation. `PWP.Post` enthält eine *POST*-Operation, wenn diese noch keine *WAITCLEAR*-Operation getriggert hat. Synchronisations-Races werden an folgenden Stellen getestet:

- Zeile 8 testet, ob bereits eine nicht verarbeitete *POST*-Operation existiert. Dann werden durch Ausführung des aktuellen Tasks die Bedingungen von Definition 11.2 direkt erfüllt. In der konkreten Implementierung wird noch der Sonderfall der redundanten *POST*-Operationen betrachtet, nach denen keine weiteren *WAITCLEAR*-Operationen mehr statt-

finden. Diese Spezialbehandlung wurde hier aus Gründen der Übersichtlichkeit weggelassen.

- Zeile 10 testet die Bedingung von Satz 1.
- Zeile 14 testet, ob zwei *WAITCLEAR*-Operationen parallel ausgeführt werden. Die erste Klausel testet direkt auf das Vorhandensein einer parallelen *WAITCLEAR*-Operation in der aktuellen Ausführungsreihenfolge. Die zweite Klausel testet, ob von der letzten bereits getriggerten *WAITCLEAR*-Operation ein Pfad zur aktuell erreichten *WAITCLEAR*-Operation existiert.

Die Zeilen 19-27 verarbeiten getriggerte *WAITCLEAR*-Operationen. Sobald alle Mitglieder einer Synchronisation (eine point-to-point Synchronisation hat nur ein Mitglied) ihre entsprechenden Flags gesetzt haben, werden happened-before-Relationen von den *POST*-Operationen zu allen aktiven *WAITCLEAR*-Operationen eingefügt (Zeile 21). Da diese *WAITCLEAR*-Operationen damit getriggert sind, wird die Ausführung der entsprechenden bisher unterbrochenen Threads wieder aufgenommen (Zeile 26). Ist die aktuell ausgeführte Operation eine *WAITCLEAR*-Operation, wird die Ausführung des aktuellen Threads dann abgebrochen (Zeile 28). Sollte die *WAITCLEAR*-Operation zu dem Zeitpunkt noch nicht getriggert sein, wird die Ausführung von einem Thread, der eine passende *POST*-Operation ausführt, wieder aufgenommen werden. Ansonsten wurde mit der Thread-Ausführung bereits implizit auf Zeile 26 fortgefahren.
Die Behandlung nicht-blockierender kollektiver Synchronisationen wurde im Listing weggelassen. Diese Operationen benötigen einen erweiterten Test auf Zeile 19, da mit der (partiellen) Fortführung der partizipierenden Threads bereits fortgefahren werden muss, wenn noch nicht alle Threads ihre entsprechende *WAITCLEAR*-Operation erreicht haben.

4.1.4 Algorithmische Komplexität

Jeder Task wird von `replay_tasks` genau einmal verarbeitet. Insgesamt wird also der Schleifenkörper $|T|$ mal ausgeführt, wobei $|T|$ die Anzahl aller Tasks ist. Folgende Stellen innerhalb des Schleifenkörpers verbrauchen mehr als konstante Zeit:

- Der rekursive Aufruf von `replay_tasks` auf Zeile 26 nimmt nur die Ausführung vorher unterbrochener Threads wieder auf. Tasks werden dadurch nicht noch einmal verarbeitet.
- Auf Zeile 5 findet eine Zuordnung von einem Event zur entsprechenden `PWP`-Datenstruktur statt. Diese Zuordnung benötigt maximal $log(|e|)$ Schritte, wobei $|e|$ die Gesamtzahl aller Events ist.
- Auf den Zeilen 10 und 14 finden Erreichbarkeitstests statt. Eine einfache Tiefensuche benötigt für diese Aufgabe $|T|$ Schritte.

Aus diesen Überlegungen ergibt sich für den Algorithmus eine Gesamtkomplexität von $\mathcal{O}(|T| * log(|e|) + |T|^2)$. Unter der Annahme $|e| \ll |T|$ kann die Komplexität zu $\mathcal{O}(|T|^2)$ vereinfacht werden.

Die Erreichbarkeitstests sind somit der laufzeitkritische Teil des Algorithmus, da diese die quadratische Komplexität verursachen. Erreichbarkeitstests über große Graphen können auf vielfältige Weise optimiert werden [YCZ10, SABW13, SM11, VCJZ14], wobei im Allgemeinen eine Indizierung der Knoten derart vorgenommen wird, dass Suchpfade abgekürzt werden. Eine grundlegende Art der Indizierung stellt dabei die *topologische Sortierung* dar. In einem topologisch sortierten azyklisch gerichteten Graph ist jedem Knoten T_n ein eindeutiger Index i_n zugeordnet, wobei $i_1 < i_2 \rightarrow T_2 \not< T_1$ gilt. Auch in einem topologisch sortierten Graphen ist die Komplexität des Erreichbarkeitstests im pathologischen Fall $\mathcal{O}(|T|)$. In der Praxis werden jedoch große Teile des Suchraums durch die Sortierung sehr schnell ausgeschlossen, so dass die Komplexität sub-linear wird. Allerdings benötigt die Berechnung eines Indexes selbst $|T|$ Berechnungsschritte. Der Aufwand würde also nur vom Erreichbarkeitstest zur Index-Berechnung verschoben werden.
Eine wichtige Erkenntnis ergibt sich an dieser Stelle aus einer möglichen Interpretation der topologischen Sortierung.

> Seien die Knoten (eines DAG) auszuführende Tasks und die Kanten repräsentieren Beschränkungen derart, dass ein Task vor einem anderen Task ausgeführt werden muss; in dieser Anwendung ist eine topologische Sortierung (der Knoten) eine zulässige Ausführungsreihenfolge der Tasks. [1]

Nach dieser Interpretation entspricht der Graph genau dem hier verwendeten Konzept des Task Graphen. Außerdem arbeitet die Funktion `replay_tasks` die Tasks in einer zulässigen Reihenfolge ab, da die Synchronisationsbeziehungen beachtet werden. *WAITCLEAR*-Operationen gelten dabei als abgearbeitet, wenn sie vollständig ausgeführt (also getriggert) und wieder verlassen wurden. Das aber bedeutet, dass die Abarbeitungsreihenfolge des Replay-Verfahrens bereits einer topologischen Sortierung entspricht. Es ist also möglich, während der modellhaften Ausführung die ausgeführten Tasks mit Hilfe eines globalen Zählers durchzunummerieren. Die Platz- und Zeitkosten dieser Nummerierung sind konstant und vernachlässigbar klein. Jeder Knoten erhält einen Integer-Wert als weiteres Datenelement und bei jeder abgeschlossenen Ausführung eines Tasks wird der globale Zähler inkrementiert und der Wert dem zuletzt ausgeführten Task zugewiesen. Damit hat die Erzeugung einer topologischen Sortierung keine Auswirkungen auf die Komplexität des Algorithmus, reduziert jedoch den Aufwand für den Erreichbarkeitstest entscheidend.
Eine weitere Optimierung des Erreichbarkeitstests nutzt die dem Task Graphen zugrundeliegende Semantik einer Aufteilung in Threads. Erreicht die Tiefensuche den Thread, in dem auch der Zielknoten liegt, so kann die Erreichbarkeit direkt getestet und damit dieser Suchzweig beendet werden. Mit Hilfe dieser beiden Optimierungen wird die praktische Gesamtkomplexität des Algorithmus sub-quadratisch bzw. quasi-linear. Eine Evaluierung dieser Komplexität wird in Abschnitt 6.3.2 (Seite 105) vorgenommen. Darüber hinaus kann die topologische Sortierung auch für weitere Aufgaben, die eine Tiefensuche benötigen (z.B. Test auf data races, siehe Abschnitt 5.2.2 (Seite 72)), verwendet werden.

[1] For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. [Rev16]

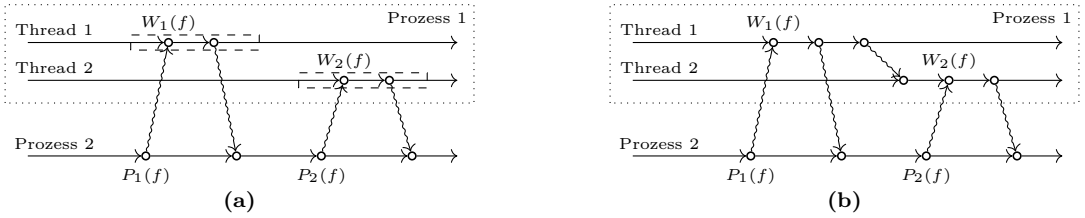


Abbildung 4.4: Heuristische Überführung von ungerichtete in gerichtete Synchronisation

4.1.5 Ungerichtete Synchronisation

Im bisher eingeführten Modell wurde ungerichtete Synchronisation nicht betrachtet. Im Zusammenhang mit dem hier entwickelten Verfahren spielt ungerichtete Synchronisation dann eine Rolle, wenn Synchronisationsoperationen innerhalb von kritischen Abschnitten aufgerufen werden. Abbildung 4.4a ist eine Erweiterung des Beispiels aus Abbildung 4.2a. In diesem Fall sind die beiden *WAITCLEAR*-Operationen W_1 und W_2 durch einen kritischen Abschnitt geschützt, der in der Abbildung durch gestrichelte Rahmen symbolisiert wird. Außerdem quittiert jeder Thread den Erhalt der Notifikation an Prozess 2. Da W_1 und W_2 nicht mehr gleichzeitig aktiv sein können, verschwindet das Synchronisations-Race. Zwar ist die Zuordnung der *POST*-Operationen zu den *WAITCLEAR*-Operationen nicht eindeutig und von der konkreten Reihenfolge der Abarbeitung der kritischen Abschnitte abhängig, jedoch wird am Ende jede *POST*-Operation genau eine *WAITCLEAR*-Operation getriggert haben. Ungerichtete Synchronisation hat also Einfluss auf das Vorhandensein von Synchronisations-Races und die Konstruktion des Task Graphen.

Anders als gerichtete Synchronisation führt ungerichtete Synchronisation nicht zu logisch zwingenden happened-before-Relationen zwischen Programmereignissen, da kritische Abschnitte in einer alternativen Programmausführung in einer anderen Reihenfolge ausgeführt werden könnten. Satz 2 gilt also nicht bei Vorhandensein von ungerichteter Synchronisation. Eine Analyse aller möglichen Ausführungen ist NP-vollständig [NM90], so dass eine skalierbare Berechnung nicht möglich ist.

In der Praxis spielt ungerichtete Synchronisation vor allem auf Thread-Level eine große Rolle. Deswegen wird in dieser Arbeit eine heuristische Erweiterung des Verfahrens zur Behandlung von ungerichteter Synchronisation verwendet. Dazu werden kritische Abschnitte in happened-before-Relationen transformiert. Unter Beachtung der tatsächlich aufgezeichneten Reihenfolge der kritischen Abschnitte wird eine happened-before-Relation vom Verlassen eines kritischen Abschnitts zum nächsten Eintritt in diesen Abschnitt in den Task Graph eingefügt. Dieser Schritt wird in das Replay-Verfahren integriert, wobei die Semantik der *lock*- und *unlock*-Operationen beachtet wird. Sind während des Replays alle Threads suspendiert, dann wird die *lock*-Operation ausgeführt, die auch in der tatsächlichen Programmausführung als erstes ausgeführt wurde. Außerdem wird eine happened-before-Relation von einer eventuell vorhergehenden *unlock*-Operation auf denselben kritischen Abschnitt zur *lock*-Operation eingefügt. Abbildung 4.4b zeigt die Überführung ungerichteter Synchronisation in das Task-Graph-Modell. Satz 1 behält in jedem Fall seine Gültigkeit. Würden die Threads den Erhalt der Notifikation nicht quittieren, so wäre $W_1 \not\prec P_2$. Das entstehende Synchronisations-Race könnte sich manifestieren, wenn P_1 und P_2 beide ausgeführt wären, bevor ein Thread seine *WAITCLEAR*-Operation betritt. In diesem Fall würde ein *POST* verlorengehen und das Programm würde nicht korrekt terminieren.

4.1.6 Weiterführende Überlegungen zum Synchronisationsmodell

Der in diesem Abschnitt entwickelte Algorithmus und das zugrundeliegende Modell liefern mehrere Beiträge zur Analyse von Synchronisationsbeziehungen in parallelen Programmen. Die einheitliche Behandlung von point-to-point und kollektiver Synchronisation ermöglicht eine Berechnung dieser Beziehungen für hybrid parallele Systeme. Durch die Einführung der *WAITCLEAR*-Operation als eine Verbindung von *WAIT* und *CLEAR* ist dieses Problem nicht mehr NP-vollständig, sondern praktisch lösbar. Der Grund für die massive Reduktion der Komplexität ist Satz 2, der so nur für Programmausführungen mit *WAITCLEAR*-Operationen gilt. Durch die Optimierung mittels topologischer Sortierung wird der Algorithmus darüber hinaus skalierbar, da er quasi-lineare Komplexität erreicht. Im Ergebnis können die logisch notwendigen Synchronisations-Beziehungen zwischen den Threads einer hybrid parallelen Programmausführung berechnet werden, indem lediglich die Programmereignisse aufgezeichnet werden. Auftretende Synchronisations-Races werden während dieser Berechnung erkannt.

In GASPI wird die *WAITCLEAR*-Operation durch die `gaspi_wait_reset`-Funktion direkt abgebildet. Damit können die Synchronisationsbeziehungen von GASPI-Programmen ohne weitere Anpassungen des Algorithmus analysiert werden. Genau so können z.B. auch `MPI_Send` und `MPI_Recv` als Ausprägungen der *POST*- bzw. *WAITCLEAR*-Operation aufgefasst werden. Der Algorithmus ist also auch für das Auffinden von message races verwendbar. Jedoch verlangt die *WAITCLEAR*-Operation keine atomare Verbindung von *WAIT* und *CLEAR*. Damit kann der Algorithmus prinzipiell auch für die Analyse genutzt werden, wenn der Programmierer die Funktionalität der *WAITCLEAR*-Operation sicherstellt. Zum Beispiel definiert der OpenSHMEM-Standard die Funktion `shmem_int_wait`. Diese Funktion wartet, bis eine Integer-Variable einen Wert ungleich eines vorgegebenen Wertes hat. Insofern entspricht diese Funktion der *WAIT*-Operation. Wenn der Programmierer sicherstellt, dass direkt nach dem Verlassen der Funktion der Wert der Variable wieder zurückgesetzt wird, so ist eine Analyse des Programms mit dem hier entwickelten Algorithmus ohne weiteres möglich. Die Bereitstellung einer Funktion `shmem_int_wait_and_clear` durch den OpenSHMEM-Standard würde zu Programmen führen, die implizit gegen Synchronsations-Races testbar sind. Daraus kann gefolgert werden, dass die während der Entwicklung von Analysewerkzeugen gewonnenen Erkenntnisse in die Entwicklung der zu analysierenden Programmiermodelle zurückfließen sollten. Insbesondere hat das Design von APIs direkten Einfluss auf die Möglichkeiten der Programmanalyse. Das GASPI-Projekt stellt dabei ein gutes Beispiel für einen entsprechenden Rückfluss dar, da die Entwickler der Analysewerkzeuge schon bei der Definition des GASPI-APIs mit einbezogen wurden. Der GASPI-Vorläufer GPI hatte keine dedizierte *WAIT*- oder *WAITCLEAR*-Operation. Stattdessen war der Programmierer darauf angewiesen, den Abschluss einer put-Operation von Hand zu überprüfen. Die automatische Erkennung und Aufzeichnung einer solchen Überprüfung ist jedoch nur schwer möglich. Deswegen wurden in GASPI Funktionen speziell für diese Aufgaben eingeführt, die direkt aufgezeichnet werden können. Die Semantik der Funktionen wurde so definiert, dass sie dem hier entwickelten Modell genügen.

Der Algorithmus wurde hier im Kontext der post-mortem-Analyse einer Programmaufzeichnung vorgestellt. Eine Adaption auf eine Online-Analyse ist jedoch ohne weiteres möglich. Da

die Tasks in jeder beliebigen gültigen Ausführungsreihenfolge abgearbeitet werden können, kann dieses auch in der Reihenfolge der eigentlichen Abarbeitung geschehen. Auf diese Weise können wesentlich längere Programmläufe analysiert werden, da bereits verarbeitete Tasks wieder gelöscht werden können. In diesem Zusammenhang stellt sich die in dieser Arbeit nicht weiter behandelte Frage, welche Tasks sicher gelöscht werden können, ohne dass die Tests auf den Zeilen 10 und 14 (`PWP[r].PrevWait` $\prec$ `T`) in Listing 4 beeinflusst werden. Die Beantwortung dieser Frage bleibt ein Thema für weiterführende Forschung.

4.2 Task Graphen für Programme mit asynchronen einseitigen Kommunikationsoperationen

Die Grundprinzipien der Modellierung einseitiger Kommunikationssysteme mithilfe von virtuellen Tasks wurden in [KKMPN13] beschrieben. Das dort vorgestellte Modell bezieht sich auf ein homogen paralleles System, in dem jeder Prozess genau einen Thread hat. In der vorliegenden Arbeit wird dieses Modell auf hybrid parallele Systeme erweitert.
Voraussetzung ist wie im vorherigen Abschnitt ein partieller Task Graph $\mathcal{P}^T = \left\langle E, \prec^T \right\rangle$. Die Menge E enthält nun alle relevanten Programmereignisse aller Threads. Relevante Programmereignisse in E sind das Aufrufen und Verlassen von Funktionen sowie direkte Speicherzugriffe. Die Relation $\prec^T$ bezeichnet wieder die direkt ableitbare sequentielle Ausführungsreihenfolge der Tasks innerhalb eines Threads.
Ziel ist die Erzeugung eines vollständigen Task Graphen $\mathcal{P} = \langle E \wedge V, \prec \rangle$. Die Menge $E \wedge V$ enthält alle lokalen und entfernten, synchronen und asynchronen Speicherzugriffe. Die Relation $\prec$ repräsentiert die logischen Abhängigkeiten dieser Zugriffe untereinander.
Die Generierung von $\mathcal{P}$ aus $\mathcal{P}^T$ geschieht in zwei Schritten. Im ersten Schritt werden die happened-before-Relationen der Threads und Prozesse untereinander berechnet. Das zugehörige Verfahren wurde im vorhergehenden Abschnitt beschrieben. Im zweiten Schritt werden dann asynchrone Speicherzugriffe zusammen mit den entsprechenden happened-before-Relationen zu den entsprechenden Kommunikationsoperationen hinzugefügt.

4.2.1 Asynchrone Ereignisse in Task Graphen

In der klassischen Sichtweise ist jeder Task Teil eines Prozesses. Das heißt, der Task wird von der Berechnungseinheit ausgeführt, auf der auch der entsprechende Thread ausgeführt wird. Die in dieser Arbeit verwendete Definition 8 abstrahiert diesen Task-Begriff. Insbesondere wird keine Zuordnung zu einem bestimmten Thread verlangt. Diese Abstraktion wird notwendig, um asynchrone Programmereignisse korrekt in einem Task Graph modellieren zu können.

Beispiel:

> Ein Blitter ist ein Coprozessor, der Speicherbereiche parallel zur auf dem Hauptprozessor stattfindenden Programmausführung kopieren kann [MDD+89]. Die Funktion `blit_copy` programmiert den Blitter mit dem zu kopierenden Speicherbereich und

startet dann den Kopiervorgang. Das Programmereignis BC bezeichnet das Betreten dieser Funktion. Die Funktion `blit_wait` wartet auf den Abschluss der aktuellen Blitteroperation. Das Programmereignis BW bezeichnet das Verlassen dieser Funktion. Der Kopiervorgang wird in zwei separate Programmereignisse zerlegt: die Operation R liest Daten aus dem Quellspeicher und die Operation W schreibt diese Daten in den Zielspeicher.

Es gilt $BC \prec R$ und $BC \prec W$, da die Lese- und die Schreiboperation von `blit_copy` angestoßen werden und also erst nach dem Betreten der Funktion BC ausgeführt werden können. Weiterhin gilt $R \prec W$ aufgrund der Datenabhängigkeit, denn nur Werte, die vorher gelesen wurden, können auch geschrieben werden. Schlussendlich gilt auch $R \prec BW$ und $W \prec BW$, da BW (also das Verlassen der Funktion `blit_wait`) erst ausgeführt wird, wenn die Kopieroperation und damit R und W vollständig abgeschlossen sind.

In Abbildung 4.5 ist der entstehende Task Graph dargestellt. Durch die transitive Reduktion wurden die Beziehungen $BC \prec W$ und $R \prec BW$ entfernt. Die hervorgehobenen Kanten verbinden die Programmereignisse, welche vom auf dem Hauptprozessor laufenden Thread ausgeführt werden. In der Abbildung repräsentiert X außerdem ein beliebiges Programmereignis, welches zwischen den Aufrufen von `blit_copy` und `blit_wait` ausgeführt wird. Parallel und asynchron zu diesem Ereignis werden die Kopieroperationen R und W ausgeführt.

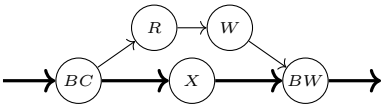


Abbildung 4.5: Task Graph einer asynchronen Kopieroperation mithilfe eines Blitters

Das Beispiel verdeutlicht eine Reihe von Abstraktionen, die bei der Modellierung von Programmausführungen mittels Task Graphen vorgenommen werden. Oft wird die komplette Ausführung einer Funktion zu einem einzelnen Task zusammengefasst. Diese Vereinfachung kann selbst dann vorgenommen werden, wenn entweder durch den Eintritt in die Funktion oder durch das Verlassen der Funktion happened-before-Relationen zu Tasks außerhalb der synchronen Prozessausführung entstehen. Werden allerdings sowohl beim Eintritt in die Funktion als auch beim Verlassen der Funktion solche happened-before-Relationen erzeugt, so müssen die beiden Ereignisse *Eintritt* und *Verlassen* auch in separaten Tasks modelliert werden. Zur Vermeidung von Zyklen ist eine solche Modellierung z.B. für kollektive Synchronisationsoperationen notwendig, die beim Eintritt eine *POST*-Operation und beim Verlassen eine *WAITCLEAR*-Operation ausführen (siehe Abschnitt 4.1.1).
Eine weitere Abstraktion stellt die aufeinanderfolgende Abbildung der Lese- und Schreiboperationen R und W dar. Tatsächlich werden die Daten von der Hardware meist nicht sofort komplett, sondern schrittweise in Blöcken kopiert. Praktisch ergäbe sich daraus eine Darstellung der Kopieroperation als $R_1 \prec W_1; R_2 \prec W_2; \ldots; R_n \prec W_n$. Je nach betrachtetem System kann

es dabei sogar zu zeitlichen Überlappungen von W_x und R_{x+1} kommen. Jedoch ist für alle in dieser Arbeit vorgestellten Anwendungen des Task-Graph-Modells eine Zusammenfassung von Kopieroperationen zu $R \prec W$ möglich.
Asynchrone Tasks sind keinem Thread zugeordnet. Solche Tasks werden im Folgenden als *virtuell* bezeichnet. Virtuelle Tasks sind nicht auf Speicherzugriffe beschränkt. Für einige Synchronisations- und Kommunikationsoperationen müssen künstliche virtuelle Tasks eingeführt werden, um die happened-before-Relationen präzise abbilden zu können. Virtuelle Tasks sind damit ein entscheidendes Mittel zur Modellierung von Systemen mit asynchroner einseitiger Kommunikation.

4.2.2 Die Modellierung von Systemen mit einseitigen Kommunikationsoperationen

Im Folgenden wird beschrieben, wie zu aufgezeichneten Programmereignissen virtuelle Tasks zugeordnet werden und so der Task Graph konstruiert wird. Dabei werden die fundamentalen einseitigen Kommunikations- und Synchronisationsoperationen der drei PGAS-APIs OpenSHMEM, GASPI und MPI-3 erörtert. API-spezifische Funktionen wie z.B. MPI-Windows werden nicht betrachtet.
Für jede beschriebene Funktion wird beispielhaft der Task Graph eines Aufrufs dargestellt. In diesen Graphen repräsentieren die fett gezeichneten Kanten die happened-before-Relationen innerhalb eines Threads. Dünn gezeichnete Kanten stellen happened-before-Relationen zu von den Funktionen erzeugten virtuellen Tasks her. Gestrichelte Kanten stehen für potentielle happened-before-Relationen zu Tasks, die von anderen Funktionen erzeugt werden.
Zusätzlichen werden formale Bildungsregeln angegeben. Die folgenden zwei Regeln werden verwendet:

Hinzufügen von Tasks $\langle X \rangle \mapsto \langle Y, Z \rangle$**:** Fügt die Menge der Tasks auf der rechten Seite zum Task Graphen hinzu. Das erste Element der Menge ersetzt dabei den aufgezeichneten Task X.

Hinzufügen von Kanten $X \to Y$**:** Fügt eine happened-before-Relationen von X nach Y ein.

Tasks, die Speicherzugriffe auf lokale Adressen repräsentieren, werden mit R_L für Lesezugriffe bzw. W_L für Schreibzugriffe bezeichnet. Speicherzugriffe auf entfernte Adressen werden mit R_R bzw. W_R bezeichnet. Tasks, die Funktionsaufrufe darstellen, werden mit dem Anfangsbuchstaben der entsprechenden Operation bzw. im allgemeinen Fall mit O abgekürzt. Funktionsargumente werden in hochgestellten Indizes notiert. Ein hochgestelltes r bezeichnet den Ziel-Rank. Weitere Argumente werden an der jeweiligen Stelle beschrieben.

Lese- und Schreiboperationen

In Tabelle 4.1 sind die Bildungsregeln für Lese- und Schreiboperationen der betrachteten Kommunikationssysteme aufgelistet. Jede dieser Operationen erzeugt einen Lese-Task R und einen Schreib-Task W auf die entsprechenden Speicherbereiche. Sei O der Aufruf der entsprechenden Operation, so gilt $O \prec R$ und $O \prec W$. Außerdem gilt aufgrund der Datenabhängigkeit $R \prec W$. Deswegen wird bei den Bildungsregeln die Beziehung $O \prec W$ weggelassen, da diese aufgrund der Transitivität der happened-before-Relation durch $O \prec R \prec W$ bereits gegeben ist und der Task Graph nur die transitive Reduktion darstellt.

Die betrachteten Kommunikationssysteme unterstützen zwei verschiedene Formen der Asynchronität. In GASPI sowie in den MPI-Funktionen `MPI_RPUT` und `MPI_RGET` sind sowohl lokale also auch entfernte Speicherzugriffe asynchron. Dementsprechend sind auch alle erzeugten Lese- und Schreib-Tasks virtuell.
Im Gegensatz dazu sind die Zugriffe auf lokalen Speicher in OpenSHMEM und in den MPI-Funktionen `MPI_PUT` sowie `MPI_GET` synchron. Im Falle von `shmem_*_get` und `MPI_GET` führt das dazu, dass die gesamte Operation synchron abläuft. Es wird also innerhalb von `shmem_*_get` gewartet, bis der entfernte Lesezugriff alle Daten übermittelt hat und diese lokal geschrieben wurden. Deswegen erzeugen diese Operationen auch keine virtuellen Tasks. `shmem_*_put` und `MPI_PUT` erzeugen einen virtuellen Schreib-Task, der die Daten asynchron auf den entfernten Speicherbereich schreibt.

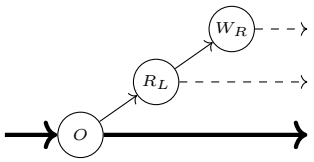
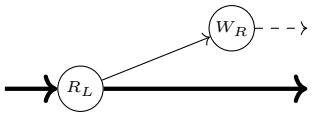
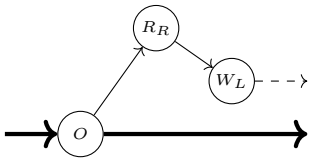
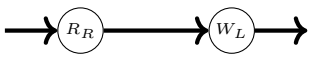
Operation	Bildungsregeln	Graph
`gaspi_write` `MPI_RPUT`	$\langle O \rangle \mapsto \langle O, R_L, W_R \rangle$ $O \to R_L$ $R_L \to W_R$	
`shmem_*_put` `MPI_PUT`	$\langle O \rangle \mapsto \langle R_L, W_R \rangle$ $R_L \to W_R$	
`gaspi_read` `MPI_RGET`	$\langle O \rangle \mapsto \langle O, R_R, W_L \rangle$ $O \to R_R$ $R_R \to W_L$	
`shmem_*_get` `MPI_GET`	$\langle O \rangle \mapsto \langle \{R_R \to W_L\} \rangle$	

Tabelle 4.1: Task Graphen für Lese- und Schreiboperationen

Lokale Synchronisationsoperationen

GASPI und MPI definieren Synchronisationsoperationen, mit denen auf den Abschluss asynchroner Zugriffe auf lokale Speicherbereiche gewartet werden kann. OpenSHMEM stellt keine derartigen Operationen zur Verfügung, da in diesem System alle lokalen Speicherzugriffe synchron zum Aufrufer sind. Ein Thread verlässt eine lokale Synchronisationsoperation erst, wenn alle dieser Operation zugeordneten lokalen asynchronen Speicherzugriffe abgeschlossen sind. WT_L bezeichnet im Folgenden das Ereignis *Verlassen der Synchronisationsoperation*. S_L bezeichnet einen lokalen asynchronen Lese- oder Schreibzugriff, es gilt also $S_L \in \{R_L, W_L\}$.

Die Zuordnung der Speicherzugriffe zu Warte-Operationen wird in GASPI und MPI unterschiedlich spezifiziert. Die MPI-Funktion `MPI_Waitall` wartet auf den Abschluss einer Menge von MPI-Requests $REQS$. Die Funktion `MPI_Wait` wartet auf den Abschluss genau eines Requests req, im Folgenden gilt für diesen Fall $REQS = \{req\}$. Diese MPI-Requests werden von den Funktionen `MPI_RPut` und `MPI_RGet` geliefert. Im Modell werden die von diesen Funktionen ausgelösten Speicherzugriffe mit dem zugehörigen Request markiert: S_L^{req}. Da req eindeutig ist, ist auch die Zuordnung von S_L^{req} zu einer Warte-Operationen WT_L^{REQS} eindeutig. Jede Warteoperation schließt alle ihren Requests $req \in REQS$ zugeordneten Lese- und Schreiboperationen S_L^{req} ab (Tabelle 4.2, 1.Zeile). Dabei ist es unerheblich, ob S_L^{req} und WT_L^{REQS} vom selben Thread aufgerufen wurden.

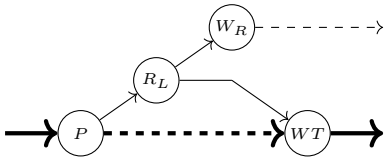
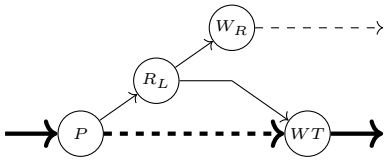
Operation	Bildungsregeln	Graph
`MPI_Wait` `MPI_Waitall`	$S_L^{req} \rightarrow WT_L^{REQS}$ mit $req \in REQS$	
`gaspi_wait`	$S_L^q \rightarrow WT_L^q$ mit $S_L^q \; before \; WT_L^q$	

Tabelle 4.2: Task Graphen für lokale Synchronisationsoperationen

GASPI stellt zur Zuordnung von Warteoperationen zu bestimmten lokalen asynchronen Speicherzugriffen nummerierte Queues bereit (siehe Abschnitt 2.7). Lese- und Schreiboperationen werden in eine vom Programmierer wählbare Queue submittiert. Ebenso erwartet `gaspi_wait` eine Queue q als Argument. Diese Operation blockiert, bis alle lokalen asynchronen Speicherzugriffe in der entsprechenden Queue abgeschlossen sind. Anders als bei MPI ist damit die Zuordnung der Speicherzugriffe zu Warteoperationen nicht mehr eindeutig.
Für homogen parallele Systeme ist die Bildungsregel $S_L^q \rightarrow WT_L^q$ für einen `gaspi_wait`-Aufruf ausreichend, wenn diese in der Reihenfolge der Thread-Abarbeitung auf alle WT_L^q-Tasks angewendet wird. Dieses Verfahren verbindet jeweils die noch nicht mit einem früheren WT_L^q-Task verbundenen asynchronen Speicherzugriffe S_L^q mit dem gerade behandelten WT_L^q-Task. Für hybrid parallele Systeme ist dieser Ansatz jedoch nicht mehr ausreichend. Abbildung 4.6 illustriert das Problem. Die zwei gezeigten Threads sind Teil eines GASPI-Prozesses. Thread 1 führt ein `gaspi_read` aus, Thread 2 ein `gaspi_wait`. Ob dieses `gaspi_wait` wartet, bis alle von `gaspi_read` gestarteten asynchronen Speicherzugriffe beendet sind, hängt von den Zeit-

punkten der Ausführung ab. Logisch notwendig beendet werden diese Speicherzugriffe nur, wenn `gaspi_wait` auch notwendigerweise nach `gaspi_read` ausgeführt wird. Es muss also eine happened-before-Relation zwischen `gaspi_read` und `gaspi_wait` existieren. In der Abbildung ist diese mögliche Beziehung gestrichelt dargestellt.

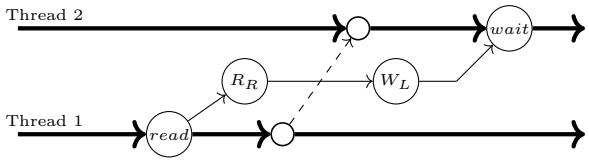


Abbildung 4.6: Synchronisation lokaler asynchroner Speicherzugriffe in einem hybrid parallelen GASPI-System

Die Bildungsregel für einen `gaspi_wait`-Aufruf darf also nur die S_L^q erfassen, die garantiert nach dem Verlassen von WT_L^q abgeschlossen sind. Weiterhin soll die Regel auch verhindern, dass von allen bereits abgeschlossenen S_L^q redundante happened-before-Relationen zu WT_L^q aufgebaut werden. Die Erfüllung der ersten Forderung hängt von der Beziehung der den Speicherzugriff auslösenden Operation $O(S_L^q)$ zu WT_L^q ab. Es gilt $S_L^q \prec WT_L^q$ genau dann, wenn $O(S_L^q) \prec WT_L^q$. Dabei ist es unerheblich, ob O und WT_L^q zum selben Thread gehören. Es muss also gelten:

$$O(S_L^q) \quad \prec \quad WT_L^q \tag{4.1}$$

Gehören O und WT_L^q zum selben Thread, dann ist die mögliche Beziehung $O \prec WT_L^q$ bereits implizit durch die Abarbeitungsreihenfolge gegeben. In homogen parallelen Systemen ist dies immer der Fall, weshalb für diese Systeme auch die vereinfachte Bildungsregel angewendet werden kann.

Zur Verhinderung redundanter happened-before-Relationen wird das eben eingeführte Kriterium auf alle Warteoperationen angewendet, die garantiert vor WT_L^q stattfinden, also auf alle $\overline{WT_L^q} \prec WT_L^q$. Sollte eine solche Operation $\overline{WT_L^q}$ existieren, für die ebenfalls $O(S_L^q) \prec \overline{WT_L^q}$ gilt, so wäre die einzufügende Relation $S_L^q \prec WT_L^q$ redundant, da bereits $S_L^q \prec \overline{WT_L^q} \prec WT_L^q$ gilt. Es muss also außerdem gelten:

$$\neg\exists\, \overline{WT_L^q}\ (\overline{WT_L^q} \prec WT_L^q \ \wedge\ O(S_L^q) \prec \overline{WT_L^q}) \tag{4.2}$$

Die Formeln 4.1 und 4.2 können in einem verallgemeinerten Prädikat $AE\ before\ E$ zusammengefasst werden, welches zu einem gegebene Programmereignis E alle garantiert vorher ausgelösten asynchronen Ereignisse AE liefert, wobei nur die asynchronen Ereignisse erfasst werden, die nicht auch schon von einem vorherigen E erfasst wurden. $O(AE)$ bezeichnet in der Definition die Operation, welche das asynchrone Ereignis auslöst.

$$AE\ before\ E \quad := \quad O(AE) \prec E \ \wedge\ \neg\exists\, \overline{E}\ (\overline{E} \prec E \ \wedge\ O(AE) \prec \overline{E}) \tag{4.3}$$

Unter Verwendung dieses Prädikats ist die vollständige, konfluente und Redundanzen vermeidende Bildungsregel für `gaspi_wait` in hybrid parallelen GASPI-Programmen in Tabelle 4.2, Zeile 2 angegeben.

Einseitige entfernte Synchronisation und Zugriffsreihenfolge

Das Konzept der einseitigen entfernten Synchronisation bzw. *remote synchronisation* wird momentan nur von GASPI und OpenSHMEM unterstützt. Eine einseitige entfernte Synchronisation liegt vor, wenn der Zielprozess einer put-Operation lokal, d.h. ohne Netzwerk-Kommunikation, auf den Abschluss des Schreibzugriffs testen bzw. warten kann.
GASPI beinhaltet den bereits beschriebenen Notifikations-Mechanismus über dedizierte Synchronisations-Flags. Die Beziehung zwischen dem Notifikations-Mechanismus und entfernten Schreiboperationen wird über die *non-overtaking*-Garantie hergestellt. Schreibt ein Prozess mittels `gaspi_write` über eine Queue Q Daten auf einen anderen Rank und sendet dieser Prozess danach eine Notifikation über Q zu diesem Rank, so ist der Erhalt der Daten auf dem Zielrank vor Erhalt der Notifikation garantiert. Notifikationen dürfen Daten, die über die gleiche Queue an den gleichen Rank gesendet werden, also nicht überholen. Wenn der sendende Prozess nach einer Menge von `gaspi_write`-Aufrufen ein `gaspi_notify` aufruft, so kann der Zielprozess durch Synchronisation mit der Notifikation feststellen, dass alle Daten dieser Menge angekommen sind. Die in der ersten Zeile von Tabelle 4.3 angegebene Bildungsregel für die Interaktion von `gaspi_write` und `gaspi_notify` beachtet die bereits im vorherigen Abschnitt erläuterten Besonderheiten hybrid paralleler Systeme und verwendet das in Formel 4.3 eingeführte Prädikat. So werden nur die `gaspi_write`-Operationen erfasst, die garantiert vor dem `gaspi_notify` ausgeführt wurden. Aus dem Task Graphen ist außerdem ersichtlich, dass auch das Senden der Notifikation selbst (W_F) asynchron zum sendenden Prozess verläuft.
Die zweite Zeile in Tabelle 4.3 stellt die Bildungsregel für entfernte Synchronisation in GASPI dar. WC ist die *WAITCLEAR*-Operation, die durch `gaspi_reset` repräsentiert wird. Das Prädikat *linked* nimmt Bezug auf das in Abschnitt 4.1.3 erläuterte Verfahren und liefert zu dem `gaspi_notify`, welches die asynchrone $POST$-Operation W_F ausgelöst hat, das zugehörige WC.

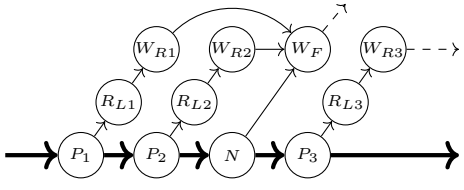
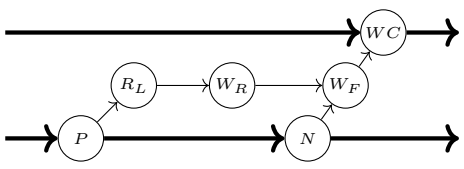
Operation	Bildungsregeln	Graph
`gaspi_notify`	$\langle N^{r,q}\rangle \mapsto \langle N^{r,q}, W_F^{r,q}\rangle$ $N^{r,q} \rightarrow W_F^{r,q}$ $\forall\ W_R^{r,q} \rightarrow W_F^{r,q}$ mit $W_R^{r,q}\ \ before\ \ N^{r,q}$	
`gaspi_reset`	$W_F \rightarrow WC$ mit $O(W_F)\ \ linked\ \ WC$	

Tabelle 4.3: Task Graphen für Zugriffsordnung und entfernte Synchronisationen in GASPI

Das OpenShmen-API stellt *wait*-Operationen auf beliebige Daten bereit. Die `shmem_*_wait`-Routinen blockieren die Ausführung solange, bis ein bestimmtes Datenwort (16 bis 64 Bit lang) eine vom Nutzer festlegbare Bedingung erfüllt. Ein solcher Test auf ein einzelnes Datenwort ist jedoch nur geeignet, um den Erhalt sehr kurzer Daten zu synchronisieren. Beim Schreiben eines größeren Datenblocks werden die Daten in mehrere Nachrichtenpakete zerlegt an den Zielprozess gesendet. Dabei kann es insbesondere auf Systemen mit *adaptive routing* dazu kommen, dass der Datenblock nicht fortlaufend auf dem Zielprozess geschrieben wird. Ein Test auf ein bestimmtes Datenwort innerhalb dieses Blocks ist dann nicht ausreichend, um den Erhalt des gesamten Blocks sicherzustellen. Deswegen sollte zur einseitigen entfernten Synchronisation eine Kombination aus `shmem_fence` und `shmem_wait` verwendet werden. Der im Task Graph in Tabelle 4.4 dargestellte Sendeprozess schreibt zuerst zwei Datenblöcke auf den Zielprozess und nach einem *fence* ein zusätzliches Datenwort. Der Zielprozess wartet auf den Empfang des einzelnen Datenwortes und kann dann aufgrund des vom Sendeprozess aufgerufenen `shmem_fence` sicher sein, dass auch der Datenblock angekommen ist. Zur Modellierung der *fence*-Operation wird für jeden Zielrank x ein künstlicher virtueller Task FO^x eingeführt. Im Gegensatz zu anderen virtuellen Tasks führt ein FO-Task keine eigentliche Operation aus, sondern stellt lediglich eine Ordnung der asynchronen Schreibzugriffe her. Alle vor FO ausgelösten asynchronen Schreibzugriffe auf einen Zielrank werden vor allen nach FO gestarteten Schreiboperationen auf denselben Rank abgeschlossen. Die Vorschrift zur Verbindung von FO mit vorherigen Schreibzugriffen ist äquivalent zu GASPI und verwendet wieder das $before$-Prädikat. Anders als in GASPI hat ein *fence* jedoch auch Einfluss auf zukünftige Schreibzugriffe. In einer weiteren Vorschrift werden

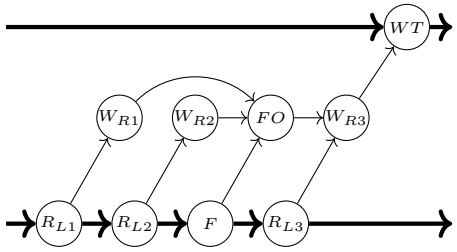
Operation	Bildungsregeln	Graph
`shmem_fence` `shmem_*_wait`	$\langle F \rangle \mapsto \left\langle F, FO^1 \dots FO^n \right\rangle$ $F \rightarrow FO^1 \dots FO^n$ $\forall\ W_R^r \rightarrow FO^r$ mit $W_R^r\ before\ F$ $\forall\ FO^r \rightarrow W_R^r$ mit $FO^r\ before\ O(W_R^r)$ $W_R \rightarrow WT$ mit $O(W_R)\ linked\ WT$	

Tabelle 4.4: Task Graph für Zugriffsordnung und entfernte Synchronisation in OpenSHMEM

deswegen happened-before-Relationen zu nachfolgenden Schreibzugriffen in den Task Graphen eingefügt. Auch hierzu wird das $before$-Prädikat verwendet, nur ist diesmal die Reihenfolge der Schreib- und Synchronisationsereignisse vertauscht.
Die von MPI bereitgestellte `MPI_Win_fence`-Funktion kann wie `shmem_fence` modelliert werden. Jedoch bietet MPI-3 noch keine entsprechende *wait*-Funktion auf Empfängerseite, weswegen eine korrekte einseitige Synchronisation wie in GASPI oder OpenSHMEM momentan schwierig zu realisieren ist. Es ist zu erwarten, dass diese Funktionalität in einer zukünftigen MPI-Version zur Verfügung stehen wird [BH15].

Kollektive Operationen

Zwar sind Barrieren und Reduktionsoperationen keine spezifischen Funktionen einseitiger Kommunikations-APIs, gleichwohl stellen sie auch bei der Programmierung mit einseitiger Kommunikation wichtige Synchronisationsmittel dar. Kollektive können unabhängig vom konkreten API immer auf die gleiche Weise modelliert werden. Dazu werden die zwei Programmereignisse *Betreten der Kollektive* BE und *Verlassen der Kollektive* BV getrennt betrachtet. Erst wenn alle beteiligten Prozesse eine Kollektive betreten haben, können diese Prozesse die Kollektive auch wieder verlassen. Daraus resultieren prozessübergreifende happened-before-Relationen $BE \prec BV$ wie sie in Tabelle 4.5 für eine Kollektive über 3 Prozesse dargestellt sind. Eine Bildungsregel ist hier nicht angegeben, da die Verbindung von BE- mit BV-Ereignissen bereits in Abschnitt 4.1 erläutert wurde. An dieser Stelle ist deswegen kein weiterer Konstruktionsschritt erforderlich.
Die Ereignisse BE und BV müssen nicht Teil desselben Funktionsaufrufs in einer Programmausführung sein. In nichtblockierenden Kollektiven bzw. *split-phase barriers* [EGCSY05] sind diese Ereignisse unterschiedlichen Funktionsaufrufen zugeordnet. Im Gegensatz zu blockierenden Barrieren können dadurch weitere Programmereignisse zwischen BE und BV auftreten. Die Modellierung ändert sich jedoch nicht, da keine asynchronen Operationen ausgeführt werden.

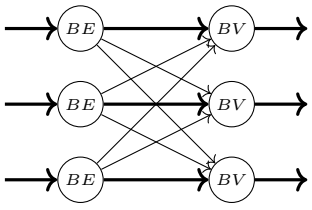
Operation	Bildungsregeln	Graph
`gaspi_barrier` `shmem_barrier`	$(BE \prec BV)$	

Tabelle 4.5: Task Graph für kollektive Synchronisationsoperationen

Reduktionsoperationen führen neben der Synchronisation noch Speicherzugriffe auf die zu reduzierenden Daten aus. Diese Speicherzugriffe müssen in den Task Graph integriert werden. Sowohl GASPI als auch OpenSHMEM stellen blockierende Reduktionsoperationen bereit. OpenSHMEM arbeitet dabei direkt auf dem Speicherbereich, der im Funktionsaufruf angegeben ist. Demzufolge finden Lese- und Schreibzugriffe auf diesen Bereich zwischen dem Betreten und dem Verlassen der Reduktion statt. Tabelle 4.6, Zeile 1 modelliert diesen Sachverhalt.
Im Gegensatz zu OpenSHMEM kopiert GASPI die zu verarbeitenden Quelldaten beim Betreten

der Reduktionsoperation. Nach Fertigstellung werden die Resultate in den Ergebnisspeicher kopiert. Lese- und Schreibzugriffe während der Bearbeitung der Reduktion finden so nur in einem internen Speicherbereich statt und müssen im Modell nicht erfasst werden. Allerdings führen die Kopieraktionen Speicherzugriffe aus, die im Modell berücksichtigt werden müssen. Der aktuelle GASPI-Standard definiert jedoch nicht, in welcher Reihenfolge das Kopieren und Kommunizieren abläuft. Das Modell in Tabelle 4.6, Zeile 2 nimmt an, dass beim Starten der Reduktion die Kopieraktion vor der ersten Kommunikation mit anderen beteiligten Prozessen stattfindet. Entsprechend wird angenommen, dass beim Verlassen der Reduktion zuerst die Kommunikation beendet wird und danach als letzter Schritt das Resultat kopiert wird. Dieses Verhalten ist intuitiv realisierbar. Eine formale Klarstellung dieser Frage durch ein zukünftiges Erratum des GASPI-Standards sollte jedoch vorgenommen werden. In dem hier gezeigten Modell sind die Speicherzugriffe einer strengeren Ordnung unterworfen. Insbesondere ist es möglich, nach dem Betreten der Reduktion auf den Quelldatenbereich zuzugreifen.

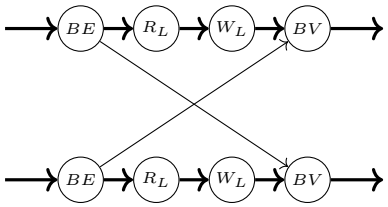
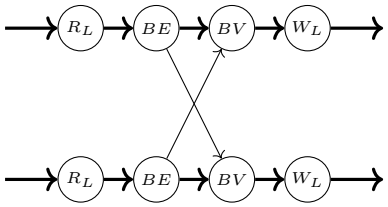
Operation	Bildungsregeln	Graph
`shmem_*_to_all`	$\langle BE \rangle \mapsto \langle \{BE \to R_L\} \rangle$ $\langle BV \rangle \mapsto \langle \{W_L \to BV\} \rangle$	
`gaspi_allreduce`	$\langle BE \rangle \mapsto \langle \{R_L \to BE\} \rangle$ $\langle BV \rangle \mapsto \langle \{BV \to W_L\} \rangle$	

Tabelle 4.6: Task Graph für kollektive Reduktionsoperationen

4.2.3 Ungerichtete Synchronisation in einseitigen Kommunikationssystemen

Ungerichtete Synchronisation mittels wechselseitigem Ausschluss wurde in den bisher eingeführten Modellen und Verfahren nicht behandelt. Wechselseitiger Ausschluss stellt in der Programmierung von Shared-Memory-Systemen ein wichtiges Synchronisationsmittel dar. Entsprechend bieten sowohl OpenSHMEM als auch MPI-3 *lock*- und *unlock*-Operationen, mit denen kritische Abschnitte definiert werden können. Beide APIs fordern, dass ein *unlock* vom selben Prozess wie das dazugehörige *lock* aufgerufen wird. Die Aufrufe werden jeweils synchron abgearbeitet und generieren keine asynchronen Ereignisse. Insofern beeinflussen diese Operationen die Konstruktion des Task Graphen nicht und auf die Angabe von Bildungsregeln wird im Folgenden verzichtet. Jedoch wird die Rolle von ungerichteter Synchronisation in einseitigen Kommunikationssystemen anhand des bisher eingeführten Modells diskutiert.

Ein kritischer Abschnitt in OpenSHMEM schützt nur die Speicherzugriffe, die garantiert vor Verlassen des Abschnitts beendet werden. Insbesondere sind asynchrone Schreiboperationen, die von `shmem_*_put` ausgelöst werden, nicht automatisch geschützt (Abbildung 4.7a). Dazu wäre es notwendig, eine happened-before-Relation vom WT-Task zur *unlock*-Operation UL zu programmieren. Lediglich die von `shmem_*_get` ausgelösten lokalen und entfernten Speicherzugriffe werden von einem kritischen Abschnitt geschützt.
MPI-3 erweitert die Semantik der *unlock*-Operation und garantiert auch die Fertigstellung asynchroner entfernter Speicherzugriffe beim Verlassen eines kritischen Abschnitts (Abbildung 4.7b). Damit wird das Problem der ungeschützten entfernten Speicherzugriffe gelöst. Allerdings wird ein kritischer Abschnitt dadurch zu einem sehr starken Synchronisationsmittel. Im Normalfall sollte ein Thread kein Interesse daran haben, wann eine einseitige Kommunikation auf der Empfängerseite abgeschlossen ist. Die Semantik der *unlock*-Operation von MPI-3 steht dieser Intention entgegen, da sie zu einer letztendlich zweiseitigen Kommunikation führt.
Offensichtlich ist eine geeignete Definition von *lock*- und *unlock*-Operationen für Systeme mit einseitiger, asynchroner Kommunikation schwierig. Die bisher definierten Ausführungsmodelle für kritische Abschnitte sind synchron zu ihren ausführenden Threads. Da auch in den Programmiermodellen für Shared-Memory-Systeme Lese- und Schreibereignisse immer synchron zu ihren ausführenden Threads sind, eignen sich kritische Abschnitte in diesen Systemen als Synchronisationsmittel. In Systemen mit asynchronen Speicherzugriffen ist dies jedoch nicht mehr der Fall. Konsequenterweise definiert der GASPI-Standard auch keine *lock*- und *unlock*-Operationen. Die entsprechende Funktionalität kann jedoch durch die Verwendung atomarer Variablen erreicht werden.

(a) `shmem_set_lock, shmem_clear_lock` (b) `MPI_Win_lock, MPI_Win_unlock`

Abbildung 4.7: Task Graphen für Lock-Operationen

4.2.4 Die Bedeutung virtueller Tasks in der Programmanalyse

In diesem Abschnitt wurde erläutert, wie Programmausführungen mit asynchronen einseitigen Kommunikationsoperationen in einem Task Graphen modelliert werden können. Anhand ausgewählter Funktionen der drei PGAS-APIs GASPI, OpenSHMEM und MPI-3 wurde gezeigt, wie sich mit Hilfe virtueller Tasks konkrete PGAS-Operationen in einen Task Graphen überführen lassen. Durch die Verwendung virtueller Task können asynchrone Programmereignisse so modelliert werden, dass ihre kausalen Beziehungen zu den übrigen Programmereignissen präzise

abgebildet werden können. Der vorgestellte Ansatz ist sowohl auf weitere Funktionen der verwendeten APIs als auch auf andere Programmiermodelle mit asynchronen Ereignissen anwendbar. Dabei zu beachtende Punkte wurden in diesem Abschnitt behandelt. So benötigen viele synchronisierende Operationen in hybrid parallelen Systemen das $before$-Prädikat. Außerdem ist für die Modellierung mancher Operationen (z.B. *fence*-Operationen) die Einführung künstlicher virtueller Programmereignisse notwendig.
Das entwickelte Modell kann als Knotenpunkt in der Methodik der Programmanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum aufgefasst werden. Auf der einen Seite abstrahiert es spezifische Eigenschaften konkreter Systeme, so dass Programmmausführungen unterschiedlicher APIs auf Task Graphen zurückgeführt werden können. Auf der anderen Seite kann das Modell als Ausgangspunkt für Analysemethoden dienen, die aus dem Task Graphen Programmeigenschaften schlussfolgern oder den Programmlauf aufbereitet präsentieren können. Mehrere sich ergebende Möglichkeiten werden im nächsten Kapitel vorgestellt.

4.3 Speicherzugriffe und sequentielle Konsistenz

In dem bisher eingeführten Modell wurden die durch PGAS-Operationen ausgelösten Speicherzugriffe (in der Folge kurz: PGAS-Speicherzugriffe) unter Beachtung aller Synchronisationsbeziehungen in den Task Graphen einer Programmausführung eingefügt. Neben diesen Speicherzugriffen führt jeder Thread auch direkte Speicherzugriffe sowohl in seinen lokalen Speicher als auch in PGAS-Segmente aus.
Direkte Speicherzugriffe sind an Programminstruktionen gekoppelt, die vom Thread ausgeführt werden. Die Programminstruktionen definieren die Zugriffsart und meistens auch die Größe des Zugriffsintervalls. Eine Ausnahme stellen auf x86-Architekturen z.B. Anweisungen mit `rep`-Präfix dar. Das Zugriffsintervall direkter Zugriffe umfasst oft nur wenige Bytes eines ununterbrochenen Adressbereichs und ist fast immer eine Zweierpotenz. Allerdings gibt es auch *gather*- und *scatter*-Instruktionen in Vektorbefehlssätzen, die mehr als nur ein Zugriffsintervall umfassen. Deswegen werden Speicherzugriffs-Ereignisse hier wie folgt definiert:

Definition 12. *Ein Speicherzugriffs-Ereignis in einem Task Graphen wird bezeichnet als* $e = (m, a)$*, wobei*

- *die Menge der Zugriffsintervalle* $m = \{I_1, I_2, \ldots, I_m\}$ *den Adress-Bereich bestimmt, auf den zugegriffen wird. Jedes Intervall ist ein fortlaufender, ununterbrochener Bereich* $I = [A_{Start}, A_{End})$ *welcher an der Startadresse* A_{Start} *beginnt und bis zur Endadresse* A_{End} *reicht.*
- *die Zugriffsart* $a \in \{READ, WRITE\}$ *bestimmt, ob lesend oder schreiben zugegriffen wird.*

Direkte Speicherzugriffs-Ereignisse werden außerdem bestimmt durch ihre Position innerhalb des ausführenden Threads. Diese Position entspricht dem logischen Zeitpunkt dieses Ereignisses. Weiterhin wird angenommen, dass direkte Speicherzugriffe keine intendierten Synchronisationsbeziehungen zu anderen Tasks erzeugen. Speicherzugriffe, durch die Synchronisationsbeziehungen entstehen, werden nicht als direkte Speicherzugriffe, sondern als ein Synchronisationsereignis des jeweiligen Typs modelliert (z.B. als eine $POST$-Operation). Aufeinanderfolgende direkte

```
1  v = 123;
2  put(&v, target_rank, ...);
```

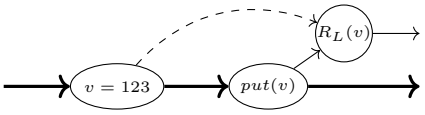


Abbildung 4.8: Datenabhängigkeiten zwischen direkten und PGAS-Speicherzugriffen

Speicherzugriffe mit gleicher Zugriffsart können je nach Anwendungsfall durch Vereinigung ihrer Zugriffsintervalle auch zu einem einzelnen Speicherzugriffs-Ereignis zusammengefasst werden.
Die synchrone Ausführung direkter Speicherzugriffe ist eine Annahme des hier eingeführten Modells, die der Intention der Ausführung eines Programms entspricht. In der Praxis muss eine geeignete PGAS-Implementierung die sequentielle Konsistenz insbesondere von direkten Speicherzugriffen und PGAS-Speicherzugriffen sicherstellen. Dabei muss der Out-of-Order-Execution moderner Prozessoren Rechnung getragen werden.
Das Beispiel in Abbildung 4.8 überträgt den Inhalt der Variablen `v` an einen Zielprozess. Offensichtlich soll der Wert 123 übertragen werden. Der nebenstehende Task Graph zeigt die zeitliche Interaktion des direkten Speicherzugriffs mit dem PGAS-Lesezugriff. Es existiert eine happened-before-Relation vom direkten Schreibzugriff zum PGAS-Zugriff, also findet das asynchrone Lesen der Variable nach dem direkten Speicherzugriff statt. Da Tasks nach Definition 8 abgeschlossen sind, ist also im Modell garantiert, dass der Lesezugriff den Wert 123 ermittelt. Ein PGAS-System sollte diesen intendierten Ablauf auch praktisch garantieren. Es ist zum Beispiel möglich, dass die Hardware Datenabhängigkeiten erkennt und entsprechend agiert.

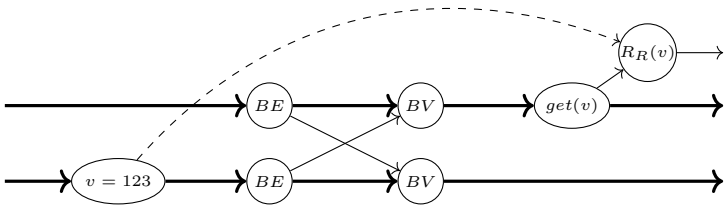


Abbildung 4.9: Datenabhängigkeiten von direkten Speicherzugriffen über Prozessgrenzen hinweg

Allerdings darf sich diese Erkennung nicht auf lediglich lokale PGAS-Zugriffe beschränken, wie der Task Graph in Abbildung 4.9 zeigt. In diesem Beispiel führt ein zweiter Prozess mittels der *get*-Operation einen entfernten Zugriff auf die Variable v durch. Auch hier existiert aufgrund der Barriere eine happened-before-Relation vom direkten Schreibzugriff zum PGAS-Lesezugriff. Kann die Hardware die Erkennung solcher Datenabhängigkeiten nicht gewährleisten, muss die PGAS-Implementierung also nicht nur PGAS-Operationen, sondern auch Synchronisationsoperationen entsprechend absichern.
Die Definition eines PGAS-APIs oder einer PGAS-Sprache sollte diesen Problemen Rechnung tragen und ein entsprechendes Speicherkonsistenzmodell bereitstellen. Für den aktuellen GASPI-Standard wurde ein solches Modell anhand der eben gezeigten Beispiele in Form eines Erratums akzeptiert [Krz15].

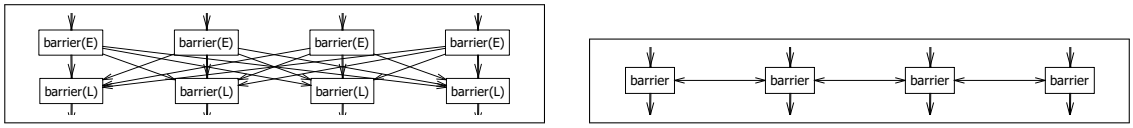
5 Anwendungskonzepte des Modells

Beauty is the ultimate defense against complexity. [Gel98]

Das im vorhergehenden Kapitel eingeführte Task-Graph-Modell der Ausführung eines PGAS-Programms bietet eine abstrakte Sicht sowohl auf die Ereignisse eines Programms als auch auf deren kausale Beziehungen untereinander. Durch die Modellierung dieser kausalen Beziehungen werden vielfältige Anwendungsmöglichkeiten erschlossen. So können gegebene parallele Programmausführungen automatisch auf wichtige Korrektheitskriterien wie z. B. das Nichtvorhandensein von unerwünschtem Nichtdeterminismus überprüft werden. Darüber hinaus bietet das Modell auch Möglichkeiten, potentielle Performance-Schwachstellen zu finden. Außerdem leistet das Modell einen wichtigen Beitrag zur Veranschaulichung und zum Verständnis des Verhaltens paralleler Programme.
Eine wichtige Eigenschaft der im Folgenden vorgestellten Anwendungskonzepte ist ihre Robustheit gegenüber der Anzahl der untersuchten Threads und der Länge des Programmlaufs. Meist führt schon eine Analyse eines kurzen Testlaufs auf einem kleinen System zu Ergebnissen. So waren die meisten im Rahmen dieser Arbeit gefundenen data races und Synchronisations-Races unabhängig von der Systemgröße eine inhärente Eigenschaft der analysierten Programme. Zwar manifestierten sich einige dieser Programmierfehler erst in lang laufenden Programmen mit großer Prozessanzahl, die logischen Ursachen waren jedoch schon durch die Analyse kleiner Testläufe erkennbar.
Die meisten der in diesem Kapitel gezeigten Abbildungen wurden mit dem im Rahmen dieser Arbeit entwickelten Werkzeug zur Analyse von GASPI-Programmen erstellt. Zur Verdeutlichung der Anwendbarkeit wurden neben Demonstrationsprogrammen auch Anwendungen aus der wissenschaftlichen Praxis untersucht.

5.1 Analyse eines Programmlaufs mittels Task Graphen

Der im vorherigen Kapitel beschriebene Task Graph bildet die logisch notwendigen happened-before-Relationen und die Einbettung asynchroner Ereignisse in den Programmlauf ab. Eine Visualisierung des Task Graphen macht damit die logischen Beziehungen sowohl zwischen den Threads als auch zwischen synchronen und asynchronen Ereignissen sichtbar. Die in dieser Arbeit verwendete Visualisierungsart nutzt eine zweidimensionale Darstellung. Eine Achse bezeichnet dabei den Programmfortschritt, während auf der orthogonalen Achse nacheinander die Threads der Prozesse angeordnet sind. Asynchrone Ereignisse werden zeitlich leicht versetzt neben die jeweils auslösenden Tasks gezeichnet. Auf diese Weise sind die sich während der Ausführung ergebenden happened-before-Relationen zwischen den Threads und die Beziehungen zu asynchronen Ereignissen im Kontext des Programmfortschritts intuitiv erkennbar. Eine solche Task-Graph-Visualisierung bietet eine wichtige Perspektive, um ein paralleles Programm zu analysieren,



(a) (b)

Abbildung 5.1: Detaillierte und vereinfachte Visualisierung kollektiver Synchronisationen

Interaktionen und Synchronisationen zu verstehen und so generiertes Wissen weitergeben zu können.

In den im Folgenden gezeigten Task Graphen verläuft die den Programmfortschritt repräsentierende Zeitachse von oben nach unten, Prozess-Ranks sind von links nach rechts aufsteigend geordnet, links beginnend mit Rank 0. Um eine kompakte Visualisierung zu erreichen, wurden verschiedene vereinfachende Darstellungsoptionen in das Werkzeug eingebaut. Der `gaspi_`-Präfix der GASPI-Funktionsnamen wird nicht mit angezeigt. Kollektive blockierende Synchronisationen, also z. B. klassische Barrieren werden wie in Abbildung 5.1b, vereinfacht dargestellt. Auf diese Weise beträgt die Anzahl der darzustellenden Pfeile pro kollektiver Operation nicht mehr $2 * P^2$, sondern nur noch P (P: Anzahl der an der Kollektive beteiligten Prozesse). GASPI-Funktionen ohne Synchronisationsbeziehungen zu anderen Programmereignissen werden gefiltert. Dazu gehören u.a. `gaspi_proc_rank` und `gaspi_segment_ptr`. Ebenfalls dazu gehören `gaspi_notify_waitsome`-Aufrufe, da diese, wie in Abschnitt 2.7 beschrieben, keine Synchronisationspunkte bilden. Diese Punkte werden immer erst mit dem Aufruf von `gaspi_notify_reset` erreicht.

5.1.1 Visualisierung von Prozessbeziehungen

Listing 5 zeigt eine GASPI-Implementierung einer broadcast-Funktion. Diese Funktion verteilt Daten von Rank 0 auf alle anderen Ranks. Die Verteilung der Daten erfolgt dabei in einem Binomialbaum, um die Kommunikationslast auf mehrere Ranks aufzuteilen. Jeder Rank außer Rank 0 muss zuerst die Daten erhalten (Zeilen 4-7). Dazu wird solange gewartet, bis die No-

```
1  function binomial_broadcast(seg_id, data_offset, data_size)
2  {
3    if (own_rank != 0)
4      gaspi_notification_t val = 0;
5      while (val == 0)
6        gaspi_waitsome(seg_id, notify_id, 1);
7        gaspi_reset(seg_id, notify_id, &val);
8
9    for_all (p : direct ascends of own_rank in binomial tree)
10     gaspi_write_notify(seg_id, data_offset,
11                        p, seg_id, data_offset,
12                        data_size,
13                        notify_id, 1,
14                        queue);
15 }
```

Listing 5: GASPI-Implementierung einer broadcast-Prozedur

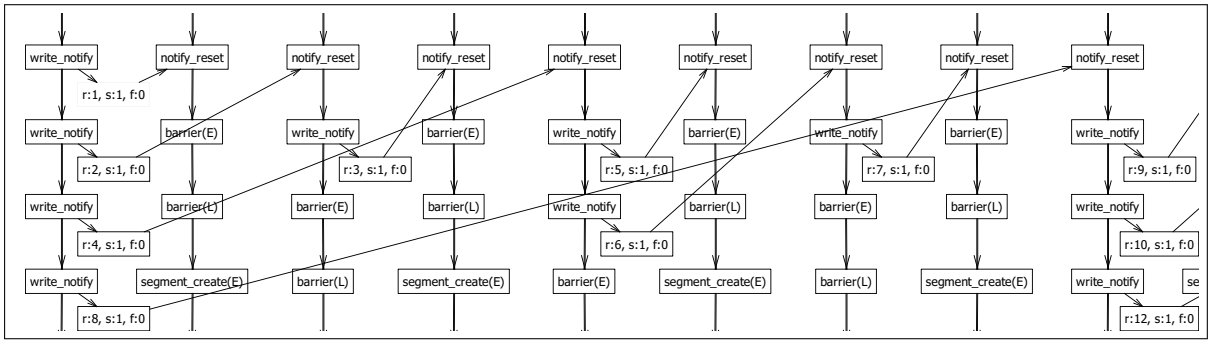


Abbildung 5.2: Asynchroner, einseitiger Broadcast implementiert als Binomialbaum

tifikation mit der ID `notify_id` im Segment `seg_id` ungleich 0 ist. Die Quelle der Daten ist dabei unerheblich. Nachdem die Daten lokal vorliegen, werden diese in der Schleife beginnend mit Zeile 9 an alle im Binomialbaum direkt untergeordneten Ranks weitergesendet.
Abbildung 5.2 zeigt einen Ausschnitt des zugehörigen Task Graphen. Rank 0 sendet die Daten an Rank 1,2,4, und 8 mittels `gaspi_write_notify`. Sobald Rank 2,4, und 8 die Daten erhalten haben, verteilen diese sie an ihre im Binomialbaum jeweils direkt folgenden Ranks. Die Struktur des Binomialbaums ist in der Task-Graph-Visualisierung schon gut erkennbar.

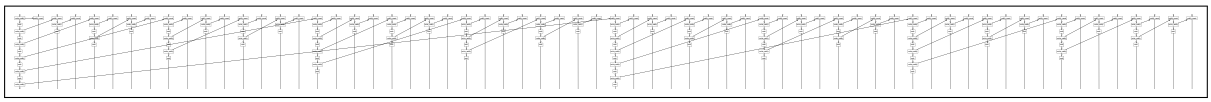


Abbildung 5.3: Kompakte Visualisierung der Synchronisationsstruktur

Eine noch kompaktere Visualisierung derselben Funktion wird in Abbildung 5.3 gezeigt. In dieser Abbildung wird auf die explizite Darstellung von asynchronen Ereignissen verzichtet. Dadurch werden $\prec$-Beziehungen zwischen den Prozessen direkt gezeichnet und Synchronisationsmuster sind auch für größere Prozessmengen ersichtlich. Abbildung 5.4 gewährt einen Eindruck auf die durch einen Butterfly-Graphen [OS98] erzeugten Prozessbeziehungen. Ein solcher Graph wird z. B. bei Fast-Fourier-Transformationen verwendet.

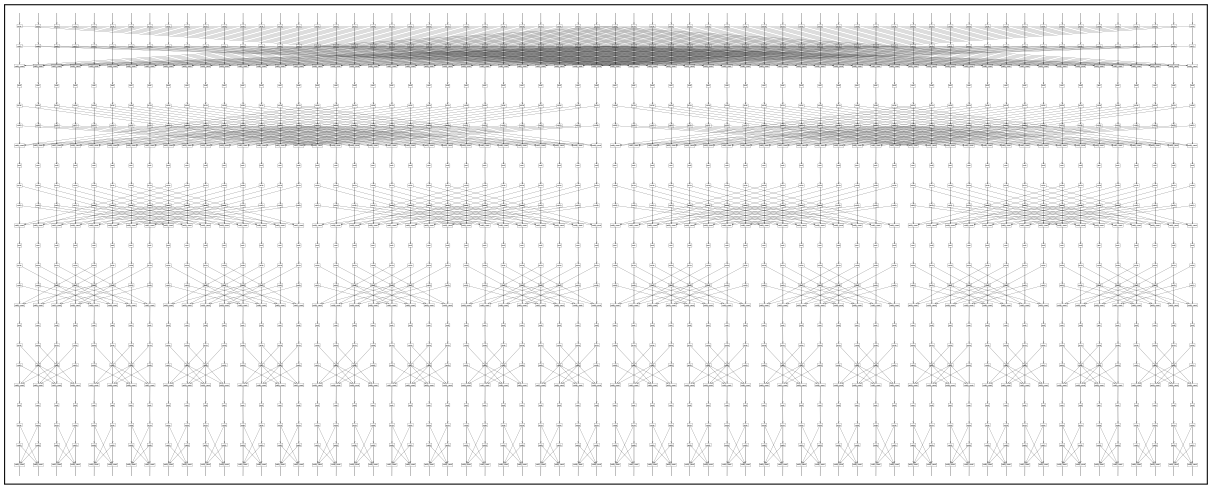


Abbildung 5.4: Überblick über einen Butterfly-Graphen von 64 Prozessen

5.1.2 Statischer Nichtdeterminismus

Task Graphen helfen nicht nur bei der Visualisierung von Prozessbeziehungen, sondern auch beim Verständnis von statischem Nichtdeterminismus. In Listing 6 ist die GASPI-Implementierung eines eindimensionalen Stencil-Codes gezeigt. Der Code benutzt zwei Segmente, die in jeder Iteration vertauscht werden. Die Berechnung ermittelt anhand der Werte im Quellsegment (`src_seg`) die Werte der nächsten Iteration und schreibt diese in das Zielsegment (`target_seg`). Zu Beginn der Iteration werden die Ränder (das Halo) des in der vorherigen Iteration berechneten Feldes an die Quellsegmente der Nachbarn übertragen. Danach kann der innere Teil des lokalen Feldes, der nicht von den Halo-Bereichen abhängt, berechnet und in das Zielsegment geschrieben werden (Zeile 16). Ab Zeile 18 wird dann auf die Übertragung der Halo-Daten von den beiden benachbarten Prozessen gewartet. Sobald ein Halo empfangen wurde, wird der lokale Randbereich, der von diesen Halo-Daten abhängt, berechnet. Dazu wird in der `while`-Schleife ab Zeile 21 auf zwei Notifikationen gewartet. Sobald eine Notifikation erhalten wurde, wird auf Zeile 25 entsprechend der Notifikations-ID entschieden, ob der obere oder untere Rand berechnet werden kann. Diese Verzweigung stellt einen statischen Nichtdeterminismus dar, denn die Reihenfolge der Berechnung des oberen und unteren Randes hängt von der tatsächlichen zeitlichen Abfolge des Erhaltes der jeweiligen Halo-Daten ab.

```
1  function stencil
2  {
3    for (i = 0; i < max; ++i)
4    {
5      src_seg = i % 2;
6      target_seg = 1 - src_seg;
7
8      gaspi_write_notify(src_seg, above_offset,
9              own_rank + 1, src_seg, below_offset, data_size,
10             notify_id, 1, queue);
11
12     gaspi_write_notify(src_seg, below_offset,
13             own_rank - 1, src_seg, above_offset, data_size,
14             notify_id + 1, 1, queue);
15
16     compute_inner_part(src_seg, target_seg);
17
18     for (k = 0; k < 2; ++k)
19       recv_id = gaspi_wait_reset(src_seg, notify_id, 2);
20
21       if (recv_id == notify_id)
22         compute_below_part(src_seg, target_seg);
23       else
24         compute_above_part(src_seg, target_seg);
25   }
26 }
```

Listing 6: 1-dimensionaler Stencil-Code

Abbildung 5.5 zeigt zwei Iterationen des Stencil-Codes in einem Programmlauf mit vier Prozessen. Das berechnete Feld repräsentiert einen Ring, so dass auch ein Rand-Austausch zwischen

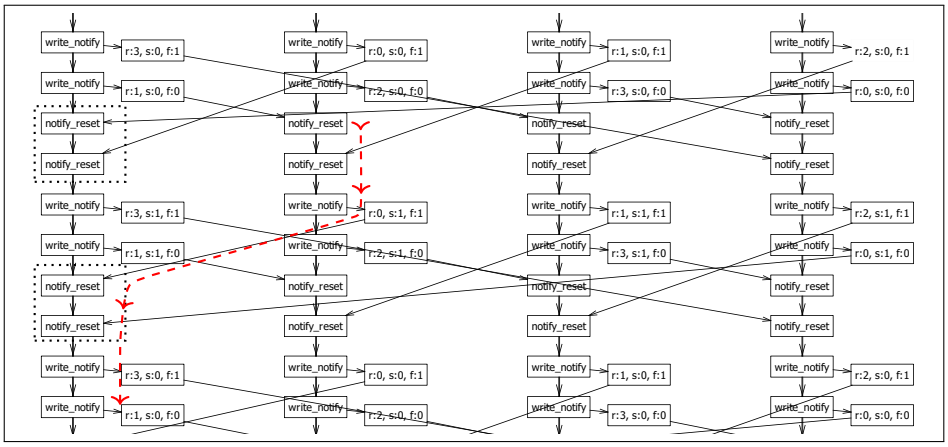


Abbildung 5.5: Synchronisationsbeziehungen in der Programmausführung eines eindimensionalen Stencil-Codes

Prozess 0 und Prozess 3 stattfindet. Der hier gezeigte Task Graph ist auf die Synchronisationsoperationen reduziert. Für die `write_notify`-Aufrufe ist deswegen nur das asynchrone Setzen der Notifikationen dargestellt, wobei `r` der Zielrank, `s` das Segment und `f` die Notifikations-ID sind. Anhand des Task Graphen können verschiedene Programmeigenschaften erkannt werden. Die zwei gepunktet umrandeten Gebiete zeigen den statischen Nichtdeterminismus. In der ersten Iteration erhält Rank 0 seine Daten zuerst von Rank 3 und danach von Rank 1. In der darauffolgenden Iteration kehrt sich die Reihenfolge um. Nun erhält Rank 0 seine Daten zuerst von Rank 1 und erst danach von Rank 3.
Die gestrichelte Linie markiert den Pfad von einer *WAITCLEAR*-Operation zur nächsten *POST*-Operation auf dieselbe Notifikation (Rank 1, Segment 0, Notifikations-ID 0). Dieser Pfad stellt eine $\prec$-Beziehung zwischen diesen Operationen dar, wie sie von Satz 1 (Seite 39) gefordert wird. Diese Beziehung wurde während der Konstruktion des Tasks Graphen durch den Replay-Algorithmus für alle *WAITCLEAR*/*POST*-Sequenzen erfolgreich getestet. Der analysierte Programmlauf enthält somit keine Synchronisations-Races.

5.1.3 Synchronisations-Races

Der Replay-Algorithmus ermittelt nicht nur die Synchronisationsbeziehungen zwischen den einzelnen Threads, sondern kann gleichzeitig zum Finden von Synchronisations-Races genutzt werden. Der Algorithmus ist in der Lage, die Art des Synchronisations-Races und die beteiligten Operationen festzustellen. Diese Informationen über gefundene Races können zum Beispiel über eine Log-Datei ausgegeben werden. Eine Visualisierung im Task Graphen eröffnet darüber hinaus die Möglichkeit, auch den Kontext eines Synchronisations-Races zu verstehen.
Listing 7 zeigt einen Ausschnitt aus einem GASPI-Programm zur Fast-Fourier-Transformation. Zu Beginn des Programmlaufs werden einige Strukturdaten mittels der schon in Listing 5 gezeigten `binomial_broadcast`-Funktion auf alle Ranks verteilt. Danach findet die eigentliche Berechnung statt. Während dieser Berechnung tauschen Prozesse Daten in Form des bereits erwähnten Butterfly-Graphen untereinander aus. Die initiale Version des Programms wurde mit einer sehr frühen Version einer GASPI-Implementierung getestet. Die berechneten Ergebnisse waren korrekt. In seltenen Fällen – häufiger in Läufen mit vielen Prozessen – lieferte das Pro-

gramm jedoch kein Ergebnis und musste von außen abgebrochen werden. Als Grund wurde ein unentdeckter Fehler in der verwendeten GASPI-Implementierung vermutet.

```
1  function fft
2  {
3    binomial_broadcast(...);
4
5    notify_id = 0;
6    queue = 0;
7    for (i = 0; i < max; ++i)
8    {
9      compute_fftw_data(...);
10
11     target_rank = compute_butterfly_target();
12     gaspi_write_notify(..., target_rank, ..., notify_id, 1, queue);
13
14     compute_twiddle_factors(...);
15
16     gaspi_wait_reset(..., notify_id, 1);
17     gaspi_wait(queue);
18   }
19 }
```

Listing 7: Programmausschnitt zur Fast-Fourier-Transformation

Dieses Programm war eines der ersten Programme, die mit dem in dieser Arbeit entwickelten Werkzeug untersucht wurden. Das eigentliche Ziel war die Visualisierung des Butterfly-Graphen (siehe Abbildung 5.4). Dabei wurde jedoch sofort ein Synchronisations-Race entdeckt. Beim ersten Analyselauf mit 4 Prozessen wurde in der Log-Datei folgender Text ausgegeben:

```
compute synchronziation connections...
notify/reset race detected: no path between a post and its previous wait
gaspi_notify and gaspi_notify_reset: target rank:3, flag id:0, segment: 3
notify/reset race detected: no path between a post and its previous wait
gaspi_notify and gaspi_notify_reset: target rank:1, flag id:0, segment: 3
error: non-triggered wait: gaspi_notify_reset
error: non-triggered wait: gaspi_notify_reset
error: non-triggered wait: gaspi_notify_reset
warning: stale notify: pgaspi_write_notify
error: unfinished barrier: gaspi_barrier(E)
```

Anhand dieser Ausgaben wird zwar klar, dass das analysierte Programm ein Synchronisations-Race enthält. Auch wird die Art des Races in Bezug auf das hier eingeführte Modell genannt. Jedoch sind der Kontext der beteiligten Operationen und insbesondere der Grund für die in der Log-Ausgabe genannten fehlenden Pfade nicht ersichtlich. Dieser Analyseschritt kann damit zwar Hinweise auf Fehler liefern, ein Ableiten von Korrekturmöglichkeiten gestaltet sich jedoch schwierig.

Mit der Task-Graph-Visualierung in Abbildung 5.6 können dagegen die Ursache und die Wirkungen des gefunden Synchronisations-Races einfach verstanden werden. Die rot markierten Knoten zeigen die an dem Synchronisations-Race beteiligten Operationen. Die ersten beiden `gaspi_notify`-Funktionen auf Prozess 0 werden in der `binomial_broadcast`-Routine aufge-

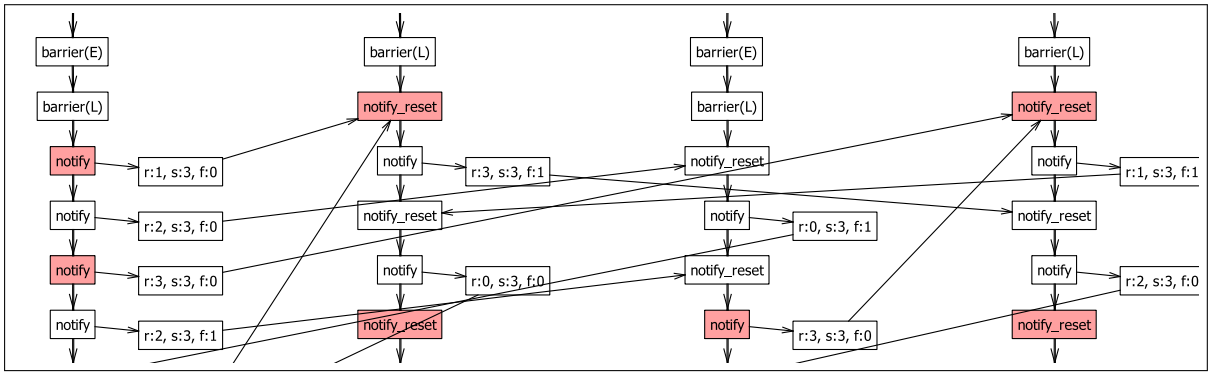


Abbildung 5.6: Synchronisations-Race

rufen. Die Operation mit Zielrank 3 kollidiert mit einer beim Datenaustausch aufgerufenen `gaspi_notify`-Funktion von Prozess 2, da nach dem in der `binomial_broadcast`-Routine ausgeführtem `gaspi_notify_reset` auf Prozess 3 und vor dem Beginn der Berechnung keine Synchronisation mit anderen Prozessen stattfindet. Eine solche Synchronisation ist algorithmisch auch gar nicht notwendig. Jedoch ist die Forderung aus Satz 1 nicht erfüllt, da beide `gaspi_notify`-Funktionen auf Prozess 3 dieselbe Notifikations-ID, somit also dasselbe Flag benutzen. Damit werden auch die beobachteten Deadlocks des Programms erklärbar. Wenn Prozess 2 bereits Daten des ersten Berechnungsschrittes überträgt, bevor Prozess 3 aufgrund von Verzögerungen die `binomial_broadcast`-Routine abgearbeitet hat, geht eine Notifikation verloren. Diese fehlt dann am Ende des Programmlaufs und die Ausführung von Prozess 3 bleibt beim Warten auf die letzte Notifikation stehen. Dieses Verhalten ist in der Task-Graph-Visualisierung sehr gut nachvollziehbar. Das zweite von Prozess 3 ausgeführte und ebenfalls markierte `gaspi_notify_reset` wird nie getriggert, denn es gibt keinen zu diesem Knoten hinführenden Pfeil.
Ein Auffinden dieses Fehlers durch eine einfache Analyse des Programmtextes durch einen Programmierer wäre sehr schwierig geworden, da die `binomial_broadcast`-Funktion und die Berechnung zu unterschiedlichen Programmmodulen gehören. Die Task-Graph-Visualisierung hat dagegen den Fehler direkt offenbart. Auch ist es mit Hilfe des in dieser Arbeit eingeführten Synchronisationsmodells möglich, die Fehlerursache präzise zu beschreiben und eine Lösung abzuleiten. Diese ist aufgrund von Satz 1 naheliegend: für den Datenaustausch während der Berechnung wird eine andere Notifikations-ID gewählt. Damit wird nicht mehr dasselbe Flag für `binomial_broadcast` und die Berechnung verwendet. Ein weiterer Testlauf zeigte dann auch keine Synchronisations-Races mehr. Ein Deadlock des Programms wurde seitdem nicht mehr beobachtet.

5.2 Die Analyse von PGAS-Programmen mittels Speicherzugriffsdiagrammen

In PGAS-Programmen bilden die Interaktionen zwischen direkten Speicherzugriffen und PGAS-Speicherzugriffen das zentrale Element der Kommunikation zwischen Prozessen. Entsprechend wichtig ist die Möglichkeit der Analyse dieser Speicherzugriffe. In diesem Abschnitt wird dazu das Konzept des Speicherzugriffsdiagramms für PGAS-Programme eingeführt.

```
1  int* Quelle = 0x10;
2  int* Ziel = 0x20;
3  for (int i=0; i<4; ++i)
4  {
5    Ziel[i] = Quelle[i];
6  }
```

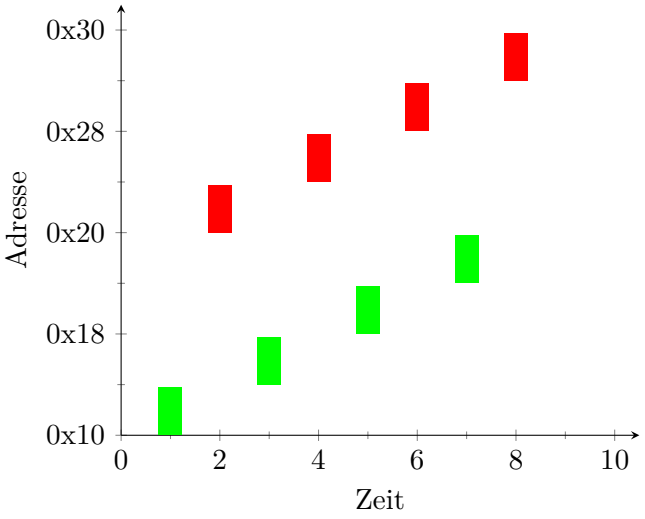
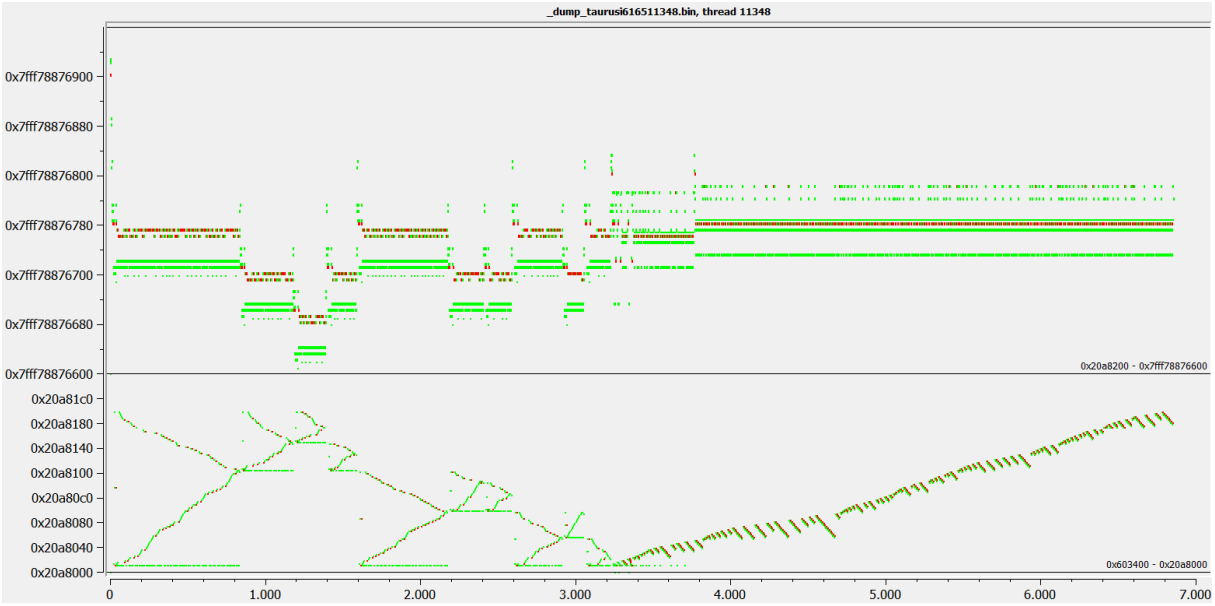
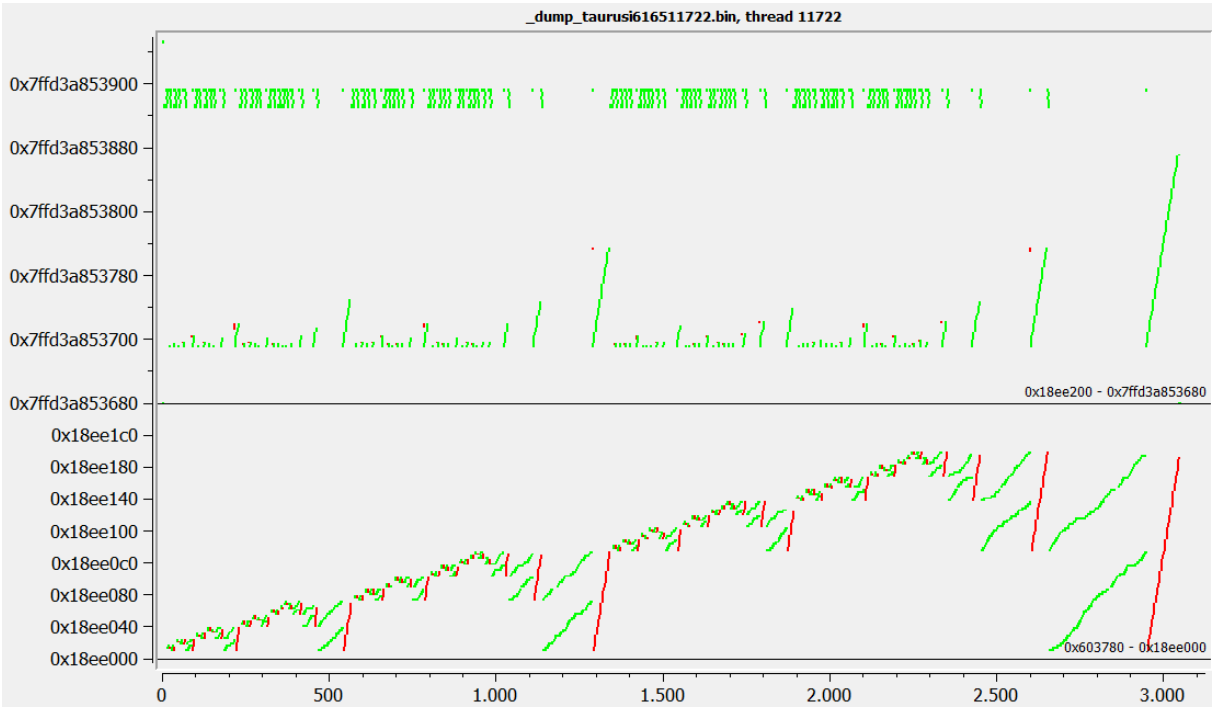


Abbildung 5.7: Die Darstellung von direkten Speicherzugriffen im Speicherzugriffsdiagramm

Ein Speicherzugriffsdiagramm im Sinne dieser Arbeit ist ein zweidimensionales kartesisches Koordinatensystem und stellt den Verlauf der Speicherzugriffe bezogen auf die Ausführung eines Threads dar. Die x-Achse bezeichnet dabei die Programmzeit, wobei sowohl ein logischer Zeitstempel als auch bei entsprechender Erfassung die Echtzeit abgetragen werden kann. Die y-Achse bezeichnet den Adressraum des Threads. In diesem Wertebereich können Speicherzugriffe stattfinden. Jeder direkte Speicherzugriff wird als senkrechte Linie dargestellt, die zum Zeitpunkt des Ereignisses das Zugriffsintervall markiert. Die Zugriffsart wird durch Farbe bestimmt: lesende Zugriffe werden grün markiert und schreibende Zugriffe rot.
Abbildung 5.7 stellt dieses Visualisierungskonzept schematisch dar. Das Speicherzugriffsdiagramm zeigt den Ablauf der Speicherzugriffe, die beim Kopieren eines Datenfeldes bestehend aus 4 Elementen in ein zweites Feld entstehen. Die Elemente sind 4 Byte große Integer. Entsprechend ist das Intervall jedes Zugriffs 4 Bytes groß. Die Quelldaten liegen an Adresse `0x10-0x20`, der Zielbereich beginnt direkt dahinter an Adresse `0x20`. Unter der Voraussetzung, dass der zur Steuerung der Schleife notwendige Zähler in einem Register gehalten wird, entsteht das gezeigte Speicherzugriffsdiagramm. In jedem Iterationsschritt wird zuerst ein Datenelement gelesen. Dieser Lesezugriff wird durch eine grüne Linie über 4 Bytes repräsentiert. Danach wird das gelesene Datenelement an die Zieladresse geschrieben. Der entsprechende Schreibzugriff wird durch eine rote Linie, die ebenfalls 4 Bytes überspannt, repräsentiert. Diese abwechselnden Lese- und Schreibzugriffe bilden im Diagramm ein für eine regelmäßig aufsteigende Iteration über Datenfelder typisches Speicherzugriffsmuster.
In einem Speicherzugriffsdiagramm können aufgezeichnete direkte Speicherzugriffe unkompliziert dargestellt werden. Eine solche Visualisierung eines Threads kann bereits wertvolle Informationen über das Programm liefern. In Abbildung 5.8 sind die Speicherzugriffsdiagramme zweier Sortierverfahren über ein Feld von 100 zufälligen Ganzzahlen gezeigt. Die aufgezeichnete C++-Standardfunktion `std::sort` verwendet das Quicksort-Verfahren. In dem gezeigten Programmlauf war das zu sortierende Datenfeld im Adressbereich von `0x20a8000` bis `0x20a8190` platziert. Der Speicherbereich ab `0x7fff78876600` wird offensichtlich von der C++-Bibliothek zur Speicherung von Zwischenresultaten verwendet. Im Bereich des Datenfeldes entstehen die



(std::sort)



(std::qsort)

Abbildung 5.8: Speicherzugriffsdiagramme von std::sort und std::qsort über 100 Ganzzahlen (compiliert mit gcc 4.9.1)

für Quicksort typischen Zugriffsmuster, wie sie auch in Abbildung 3.1 (Seite 30) erkennbar sind. Die Funktion `std::qsort` basiert dem Namen nach ebenfalls auf dem Quicksort-Verfahren. Jedoch wird dieses Verfahren vom C++-Standard nicht garantiert [ISO12]. Tatsächlich zeigt das Speicherzugriffsdiagramm von `std::qsort` deutlich andere Zugriffsmuster als `std::sort`. Eine daraufhin erfolgte eingehendere Untersuchung ergab, dass der verwendete Compiler (gcc 4.9.1) für Datenfelder dieser Größe das Mergesort-Verfahren verwendet. Im Speicherzugriffsdiagramm von `std::qsort` ist dieses Datenfeld im Adressbereich von `0x18ee000` bis `0x18ee190` platziert. Außerdem wird ein an Adresse `0x7ffd3a853680` beginnender Zwischenspeicher verwendet. Anhand der Zeitstempel auf der x-Achse kann festgestellt werden, dass das Mergesort-Verfahren für die Sortierung des gegebenen Feldes wesentlich weniger Speicherzugriffe benötigt. Gut erkennbar sind außerdem die unterschiedlichen Teile-und-herrsche-Strategien der beiden Verfahren. Speicherzugriffsdiagramme bieten damit die Möglichkeit, Algorithmen zu veranschaulichen.
In beiden Abbildungen wird der Adressraum gefaltet dargestellt, indem jeweils die Adressbereiche zwischen `0x20a8190` und `0x7fff78876600` bzw. `0x18ee190` und `0x7ffd3a853680` ausgeblendet werden. In diesen Bereichen finden keine Speicherzugriffe statt. Eine horizontale schwarze Linie stellt die Position einer Faltung dar. Diese Linie ist am rechten Ende mit dem ausgeblendeten Adressbereich beschriftet. Sollten Speicherzugriffe im gefalteten Bereich existieren, so werden diese auf der Faltungslinie gezeichnet. Die später in diesem Abschnitt erläuterte Abbildung 5.11 beinhaltet ein weiteres Beispiel dieser Darstellung.
Das Prinzip der Faltung ist für eine praktikable Darstellung von Speicherzugriffen essentiell. Die Zusammenfassung der Adressen aller Speicherzugriffe einer Programmausführung ergibt meist ein sehr großes Zugriffsintervall. In diesem Intervall gibt es aber systembedingt sehr viele Lücken, in denen überhaupt keine Speicherzugriffe stattfinden (z. B. der Bereich zwischen Stack und Heap). Solche Lücken können automatisch gefaltet werden. Darüber hinaus kann eine interaktive Faltung von Adressbereichen, in denen für die Programmanalyse weniger interessante Dinge stattfinden, wichtige Details hervorheben. Ein Zoomen in einen solchen Bereich ist zwar auch möglich, allerdings geht dann die Übersicht über den zeitlichen Kontext der untersuchten Speicherzugriffe zu Zugriffen in anderen Adressbereichen verloren.

5.2.1 Die Darstellung von PGAS-Speicherzugriffen in Speicherzugriffsdiagrammen

Der Wertebereich eines Speicherzugriffsdiagramms eines zu einem PGAS-Prozess gehörenden Threads umfasst nur die direkt vom Thread zugreifbaren Adressen. Damit sind zwar PGAS-Segmente enthalten, die in der eigenen Partition liegen, nicht jedoch die Partitionen anderer PGAS-Prozesse. Die Speicherzugriffsdiagramme zweier Threads ein und desselben PGAS-Prozesses zeigen also denselben Adressraum an. Demgegenüber zeigen die Speicherzugriffsdiagramme von Threads unterschiedlicher PGAS-Prozesse auch unterschiedliche Adressräume.
Wie direkte Speicherzugriffe haben auch PGAS-Speicherzugriffe ein Adressintervall und eine Zugriffsart. Im Gegensatz zu direkten Speicherzugriffen hat ein PGAS-Speicherzugriff jedoch ein potentiell sehr großes Adressintervall. Ein Zugriff auf einzelne Datenwörter innerhalb des Intervalls wird in dem hier eingeführten Modell nicht abgebildet. Stattdessen ist jedes einen PGAS-Speicherzugriff repräsentierende Programmereignis immer ein Zugriff auf das komplette Adressintervall. Dieses Intervall wird direkt auf die y-Achse des Diagramms abgetragen, wenn der

Zugriff im Adressraum des Threads liegt. PGAS-Speicherzugriffe außerhalb dieses Adressraums werden nicht gezeichnet.
Die Darstellung eines PGAS-Speicherzugriffs in einem Speicherzugriffsdiagramm bedingt eine zeitliche Einbettung dieses Zugriffs in den Verlauf des dargestellten Threads. Es muss also eine x-Koordinate gefunden werden, an deren Position das Adressintervall des Zugriffs gezeichnet werden kann. PGAS-Speicherzugriffe finden jedoch meist asynchron zur Thread-Ausführung statt. Damit ist das Finden eines genauen Ausführungszeitpunktes für einen solchen Zugriff bezogen auf den Zugriffsverlauf des im Diagramm dargestellten Threads nicht möglich. Der Zugriff findet stattdessen aus Sicht des Threads in einem Zeitintervall statt. Wenn eine vom Thread ausgeführte Operation den PGAS-Speicherzugriff initiiert, so entspricht der Startzeitpunkt des Zugriffs dem Zeitpunkt der Operation. Ab diesem Moment wird auf eine nicht näher bestimmte Weise auf das durch den Speicherzugriff definierte Adressintervall zugegriffen. Der Endzeitpunkt des Zugriffs wird definiert durch die Operation, die den PGAS-Speicherzugriff mit dem angezeigten Thread synchronisiert. Über den gesamten Verlauf vom Start- bis zum Endzeitpunkt können asynchron Zugriffe auf das gesamte Adressintervall stattfinden. Erst nach dem Endzeitpunkt sind alle Zugriffe verlässlich beendet. Damit kann ein PGAS-Speicherzugriff in einem Speicherzugriffsdiagramm nicht als genau ein Zugriff zu genau einem bestimmten Zeitpunkt dargestellt werden. Stattdessen kann sowohl in x- als auch in y-Richtung nur ein Bereich angegeben werden, in dem der Zugriff stattfindet. Dieser Bereich hat die Form eines Rechtecks, dessen Grenzen in x-Richtung von den Start- und Endzeitpunkten und in y-Richtung vom Adressintervall bestimmt werden. Wie im Task Graph Modell wird ein PGAS-Speicherzugriff in einem Speicherzugriffsdiagramm also auch immer als genau ein Ereignis gezeichnet.

```
1   int* segment = seg_ptr();
2   for (int i = 0; i < 4; ++i) segment[i + 4] = 0;
3
4   gaspi_write(/*source seg=*/0, /*offset=*/0x10,
5               /*target rank=*/1, /*seg=*/0, /*offset=*/0x10,
6               /*data size=*/0x10, /*queue=*/0);
7
8   for (int i = 0; i < 4; ++i) segment[i] = 1;
9
10  gaspi_wait(/*queue=*/0);
11
12  for (int i = 0; i < 4; ++i) printf (segment[i]);
```

Listing 8: Ein GASPI-Programm mit einem asynchronen lesenden Zugriff auf das lokale Segment

In Abbildung 5.9 wird die Darstellung eines asynchronen lokalen PGAS-Zugriffs demonstriert, der Teil des in Listing 8 gezeigten Programms ist. Das Speicherzugriffsdiagramm visualisiert den Verlauf der Speicherzugriffe für den einzigen Thread von Prozess 0. Es wird also der Adressraum von Rank 0 dargestellt. Die Adressen sind dabei auf den Start des Segmentes normiert. Der Thread initiiert einen asynchronen Lesezugriff auf das lokale Segment durch Aufruf von `gaspi_write`. Außerdem wird ein Schreibzugriff auf das entfernte Segment von Rank 1 ausgelöst. Während der Ausführung der durch `gaspi_write` gestarteten put-Operation werden wei-

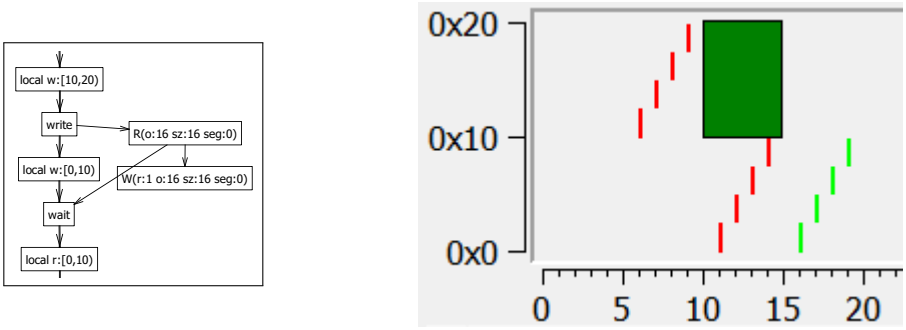


Abbildung 5.9: Der Task Graph und das Speicherzugriffsdiagramm des lokalen Segmentes von Listing 8

tere Daten im Segment an Position 0 bis 0x10 geschrieben. Danach wird der lokale asynchrone Lesezugriff durch Aufruf vom `gaspi_wait` mit dem Thread synchronisiert.

Im Speicherzugriffsdiagramm wird der asynchrone Lesezugriff als dunkelgrünes Rechteck dargestellt. Dieses Rechteck symbolisiert, dass während der gesamten Ausführung des Threads beginnend mit `gaspi_write` und bis zu `gaspi_wait` und in dieser Zeit über das gesamte Adressintervall beginnend an Offset 0x10 bis zu Offset 0x20 Lesezugriffe im Bereich des Rechtecks stattfinden. Für den PGAS-Programmierer ist diese abstrahierende Darstellung passend, da die Semantik von PGAS-Operationen auch nur genau diese Zusicherungen geben. Deutlich wird, dass die schreibenden Zugriffe auf den unteren Teil des Segmentes (Zeile 8 in Listing 8) parallel zu dem asynchronen PGAS-Zugriff stattfinden. Ebenfalls klar erkennbar ist, dass die anderen lokalen Zugriffe auf den Zeilen 2 & 12 vor bzw. nach diesem PGAS-Zugriff stattfinden. Nicht dargestellt im Speicherzugriffsdiagramm ist der ebenfalls von `gaspi_write` gestartet asynchrone Schreibzugriff. Das Ziel dieses Zugriffs ist der Adressraum von Rank 1, der nicht zum Wertebereich des gezeigten Speicherzugriffsdiagramms gehört.

Die Ermittlung des Zeitintervalls, in dem ein PGAS-Speicherzugriff aus Sicht eines Threads stattfindet, kann im Task-Graph-Modell mit einer allgemeinen Berechnungsvorschrift formuliert werden. Die gesuchten Start- und Endzeiten seien t_S und t_E. Der PGAS-Speicherzugriff sei das Ereignis p im Task Graph. Der Thread, dessen Speicherzugriffsdiagramm angezeigt wird, besteht aus den fortlaufenden Programmereignissen $e_1, e_2, \ldots e_n$, wobei der Index den Zeitstempel angibt. Dann ist der Zeitstempel i des letzten Ereignisses e_i, für das noch $e_i \prec p$ gilt, die Startzeit. Entsprechend ist der Zeitstempel j des ersten Ereignisses e_j, für das $p \prec e_j$ gilt, die Endzeit. Formal gilt also:

$$t_S = \max(0, \forall i : e_i \prec p) \tag{5.1}$$

$$t_E = \min(n, \forall j : p \prec e_j) \tag{5.2}$$

Kann keine Startzeit gefunden werden, weil kein Ereignis e existiert, für das $e \prec p$ gilt, so beginnt der Speicherzugriff bereits mit dem Start des Threads, es gilt also $t_S = 0$. Gibt es kein Ereignis e, für welches $p \prec e$ gilt, so findet der Speicherzugriff potentiell während der gesamten Restlaufzeit des Threads statt und endet also erst mit e_n.

Die Berechnung des Zeitintervalls ermittelt Zeitpunkte, die den Tasks des Threads zugeordnet

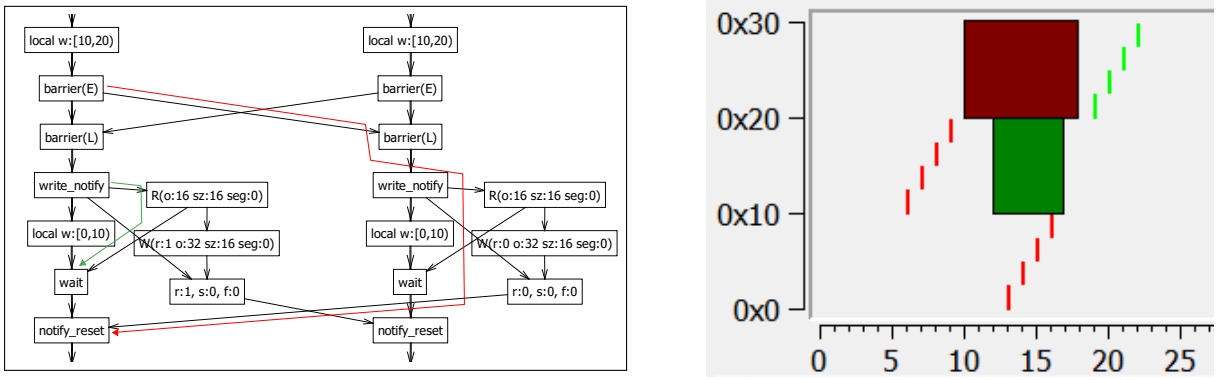


Abbildung 5.10: Darstellung eines lokalen PGAS-Lesezugriffs und eines entfernten PGAS-Schreibzugriffs im Speicherzugriffsdiagramm von Rank 0

sind. Eine genauere Bestimmung dieser Zeitpunkte wird nicht vorgenommen. Insbesondere wird nicht festgelegt, ob ein tatsächlicher Zeitpunkt bereits mit Start des Tasks oder erst mit dessen Beendigung erreicht ist. Diese Vereinfachung ist möglich, weil startende bzw. synchronisierende Tasks selbst keine Speicherzugriffe in das PGAS-Segment vornehmen. PGAS-Operationen wie zum Beispiel kollektive Reduktionen (Tabelle 4.6, Seite 55), die sowohl Speicherzugriffe als auch Synchronisationen durchführen, müssen im Modell in mehrere Tasks aufgeteilt werden.

Mit Hilfe der soeben eingeführten Formeln ist es somit möglich, sowohl lokale als auch entfernte PGAS-Speicherzugriffe auf den Speicher eines Ranks in einem Speicherzugriffsdiagramm darzustellen. Abbildung 5.10 zeigt einen entfernten PGAS-Speicherzugriff auf das Speichersegment von Rank 0. Rank 0 und Rank 1 synchronisieren sich zuerst mittels einer Barriere. Danach schreibt jeder Rank in das Segment des jeweils anderen Ranks Daten an Offset 0x10. Aus Sicht des auf Rank 0 laufenden Threads kann das Schreiben durch Rank 1 in das Segment von Rank 0 frühestens mit dem eigenen Betreten der Barriere beginnen, denn diese ist das letzte Ereignis, welches auf jeden Fall vor dem durch das `gaspi_write_notify` auf Rank 1 ausgelösten Schreibzugriff kommt. Das Betreten der Barriere (`barrier(E)`) hat den Zeitstempel 10. Beendet wird der Schreibzugriff dann durch die lokale Synchronisation per Notifikation. Diese zeigt an, dass die Daten vollständig erhalten wurden und der Schreibzugriff somit beendet ist. Der entsprechende `gaspi_notify_reset`-Aufruf hat den Zeitstempel 18. Zwischen diesen zwei Ereignissen finden genau 7 weitere Ereignisse statt: 3 Funktionsaufrufe und 4 direkte Speicherzugriffe. Die direkten Speicherzugriffe sind im Speicherzugriffsdiagramm gut als parallel zum PGAS-Zugriff ablaufende Schreibzugriffe zu erkennen. Im Task Graph markiert der rote Pfeil den Pfad vom Start bis zum Ende des Schreibzugriffs.

Im Speicherzugriffsdiagramm wird ebenfalls der lokale asynchrone Lesezugriff dargestellt. Dieser als dunkelgrünes Rechteck dargestellte Zugriff wird durch das vom Thread auf Rank 0 ausgeführte `gaspi_write_notify` ausgelöst. Der Zugriff startet später und ist aufgrund des `gaspi_wait`-Aufrufs auch schon vor dem entfernten Schreibzugriff beendet. Wäre die Reihenfolge der Aufrufe von `gaspi_wait` und `gaspi_notify_reset` vertauscht, so würde der Lesezugriff auch erst nach dem Schreibzugriff beendet werden. Im Task Graph markiert der grüne Pfeil den Pfad vom Start bis zum Ende des Lesezugriffs.

Abbildung 5.11 zeigt einen Ausschnitt des Speicherzugriffsdiagramms eines Threads des Stencil-

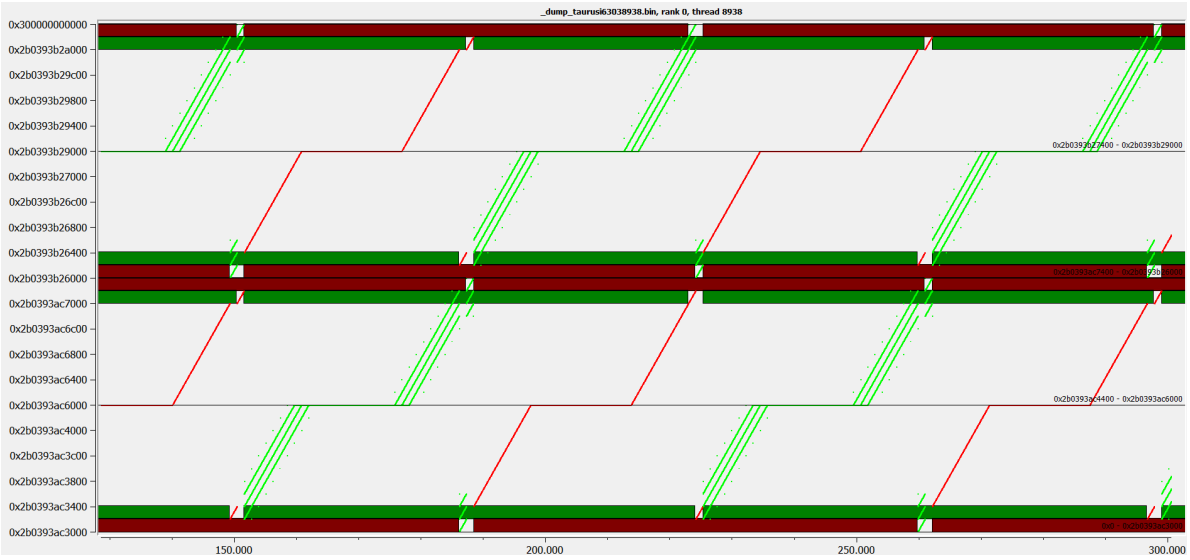


Abbildung 5.11: Faltung des dargestellten Adressraumes für große Datenmengen am Beispiel des Stencil-Programms von Listing 6

Programms (Listing 6) über ein lokales Datenfeld von $128 * 128$ Zellen. Das Double-Buffering wird mit Hilfe zweier Segmente implementiert, die insgesamt einen sehr großen Adressraum umfassen. Allerdings finden im Bereich zwischen den Segmenten keine Speicherzugriffe statt. Dieser Bereich an Position `0x2b0393b26000` wird automatisch gefaltet. Darüber hinaus wurden die Adressbereiche an `0x2b0393ac6000` und `0x2b0393b29000` interaktiv gefaltet. Damit werden die Adressräume für die rein lokale Berechnung reduziert dargestellt und die direkten und die PGAS-Speicherzugriffe in den Halo-Bereichen sind besser erkennbar.
Ein Speicherzugriffsdiagramm ist also geeignet, um die zeitlichen Abfolge der einzelnen PGAS-Speicherzugriffe und ihre logische Einbettung in den Thread-Ablauf zu erkennen. Diese Darstellungsart konnte mit Hilfe des im vorherigen Kapitel eingeführten Task-Graph-Modells entwickelt werden. Die Modellierung von PGAS-Speicherzugriffen als virtuelle Tasks macht deren Erfassung als eigenständige, parallel zur Ausführung des Programms ablaufende Ereignisse möglich. Die präzise zeitliche Einbettung in den Programmablauf und damit eine entsprechende Darstellung kann nur erfolgen, weil auch die Synchronisationsbeziehungen in Bezug auf ihre logische Notwendigkeit berechnet werden können.

5.2.2 Speicherzugriffsdiagramme und Data Races

Eine formale Regel zur Erkennung von data races in einem Task Graphen kann unter Anwendung der Kriterien von Definition 6 mit den in Definition 12 eingeführten Bezeichnern direkt formuliert werden:

Korollar 1. *Seien $e_1 = (m_1, a_1)$ und $e_2 = (m_2, a_2)$ zwei Zugriffsereignisse in einem Task Graphen T. Ein data race zwischen e_1 und e_2 existiert genau dann, wenn*

$$m_1 \cap m_2 \neq \emptyset \wedge (a_1 = WRITE \vee a_2 = WRITE) \wedge e_1 \not\prec e_2 \wedge e_2 \not\prec e_1.$$

Die Ermittlung von data races in einem Task Graphen geschieht also durch den paarweisen Test

von Speicherzugriffs-Ereignissen auf die Erfüllung der Kriterien von Korollar 1. Ein solches Vorgehen hätte mindestens quadratische Komplexität bezogen auf die Anzahl der Zugriffsereignisse. Hinzu kommen die Tests $e_x \not\prec e_y$, die durch die topologische Sortierung zwar in sublinearer Zeit, jedoch nicht in konstanter Zeit durchgeführt werden können. Für eine praktikable Korrektheitsanalyse ist es also notwendig, die Anzahl der Vergleiche zu minimieren.
Zunächst können zwei verschiedene Analysebereiche unterschieden werden. Der erste Bereich ist die Ermittlung von data races zwischen direkten Speicherzugriffen innerhalb eines Prozesses. Für diesen Bereich existieren bereits eine ganze Reihe von Algorithmen und Werkzeugen. Deswegen wird in dieser Arbeit nur der zweite Analysebereich betrachtet: die Ermittlung von data races, an denen PGAS-Zugriffe beteiligt sind. Für diesen Bereich können mehrere Einschränkungen vorgenommen werden. Erstens muss mindestens ein Zugriffsereignis ein PGAS-Speicherzugriff sein. Direkte Speicherzugriffe von Threads zweier verschiedener Prozesse können kein data race bilden, da für den Zugriff auf den Speicher eines anderen Prozesses immer eine PGAS-Operation notwendig ist. Zweitens müssen direkte Speicherzugriffe immer auf Adressen im gemeinsam genutzten Adressraum des jeweiligen Prozesses zugreifen. Zugriffe in den privaten Adressbereich können keine data races mit PGAS-Speicherzugriffen bilden, da PGAS-Operationen auf diesen Adressbereich keinen Zugriff haben. Bei der data-race-Analyse von GASPI-Programmen müssen also nur die direkten Speicherzugriffe erfasst und analysiert werden, deren Adresse in einem GASPI-Segment liegt.
Weiterhin können direkt aufeinanderfolgende direkte Speicherzugriffe zu einem einzelnen Ereignis zusammengefasst und damit die Anzahl der zu analysierenden Ereignisse erheblich reduziert werden. Dabei werden die Zugriffsintervalle der einzelnen Zugriffe zu einer Menge von Intervallen vereinigt. Da diese Menge unabhängig von der Ausführungsreihenfolge sortiert werden kann, ist der Test $m_x \cap m_y \neq \emptyset$ sehr schnell möglich. Zweckmäßigerweise werden in einem solchen Ereignis zwei Intervall-Mengen gespeichert:

1. die Intervalle aller Schreibzugriffe. Diese Menge wird gegen alle asynchronen Lesezugriffe getestet.

2. die Intervalle der Vereinigung aller Schreib- und Lesezugriffe. Diese Menge wird gegen alle asynchronen Schreibzugriffe getestet.

Dieses Vorgehen spart für asynchrone Schreibzugriffe einen zusätzlichen Test gegen die in der ersten Intervall-Menge gespeicherten lokalen Schreibzugriffe.
Nach dieser Reduktion der Anzahl der Ereignisse müssen die folgenden zwei Testläufe durchgeführt werden:

1. für jeden Thread der paarweise Test direkter Speicherzugriffe mit allen PGAS-Speicherzugriffen auf den Rank des Threads, und

2. für jeden Prozess der paarweise Test aller PGAS-Speicherzugriffe.

Zur weiteren Reduktion der Anzahl der Tests können die Formeln 5.1 und 5.2 verwendet werden. Denn nur Speicherzugriffe, deren Zeitintervalle sich überlagern, werden potentiell parallel zueinander ausgeführt und können also ein data race konstituieren. Für direkte Speicherzugriffe gilt folgende Beziehung:

Satz 3. *Sei e_t ein direkter Speicherzugriff mit Zeitstempel t und sei p ein PGAS-Speicherzugriff im Zeitintervall der Ereignisse (e_s, e_f) des Threads. e_t und p werden genau dann parallel ausgeführt, wenn $s < t < f$.*

Beweis. Zur Vereinfachung wird angenommen, dass t, s und f paarweise voneinander verschieden sind, da direkte Speicherzugriffe nicht gleichzeitig das Zeitintervall eines PGAS-Speicherzugriffs begrenzen können. Dann können drei Fälle unterschieden werden:

1. $t < s$: Dann gilt auch $e_t \prec e_s$, da beide Ereignisse Teil desselben Threads sind und innerhalb eines Threads immer $e_x \prec e_y$ für alle $x < y$ gilt. Da $e_s \prec p$ aufgrund von Formel 5.1, gilt auch $e_t \prec p$. e_t und p werden also nicht parallel ausgeführt.

2. $f < t$: Dann gilt $e_f \prec e_t$ analog zu Fall 1. Da $p \prec e_f$ aufgrund von Formel 5.2, gilt auch $p \prec e_t$. e_t und p werden also nicht parallel ausgeführt.

3. $s < t < f$: Angenommen, p und e_t werden nicht parallel ausgeführt. Dann muss entweder eine Beziehung $e_t \prec p$ oder $p \prec e_t$ existieren. Wenn $e_t \prec p$, dann muss lt. Formel 5.1 ein $s' > t$ existieren, welches das letzte Ereignis darstellt, von dem p aus noch erreichbar ist. Das widerspricht aber der Annahme $s < t$. Umgekehrt kann $p \prec e_t$ nur gelten, wenn lt. Formel 5.2 ein $f' < t$ existiert, was wiederum der Annahme $t < f$ widerspricht.

□

Für den Vergleich zweier PGAS-Zugriffe miteinander kann nur eine schwächere Regel aufgestellt werden, da eine Überlagerung der Zeitintervalle bezogen auf einen Thread nicht notwendigerweise auch bedeutet, dass beide Zugriffe parallel zueinander ausgeführt werden.

Satz 4. *Seien p_1 und p_2 zwei PGAS-Speicherzugriffe, die in einem beliebigen Thread im Zeitintervall der Ereignisse (e_{s1}, e_{f1}) bzw. (e_{s2}, e_{f2}) stattfinden. p_1 und p_2 werden nicht parallel ausgeführt, wenn sich die Intervalle nicht überlagern.*

Beweis. Wenn sich die Zeitintervalle nicht überlagern, dann kann o.B.d.A. angenommen werden, dass $f1 <= s2$ gilt. Dann gilt auch $e_{f1} \prec e_{s2}$, da beide Ereignisse vom selben Thread ausgeführt werden. Da außerdem $p_1 \prec e_{f1}$ (5.2) und $e_{s2} \prec p_2$ (5.1), gilt auch $p_1 \prec p_2$. p_1 und p_2 können nicht parallel ausgeführt werden. □

Überlagern sich die Zeitintervalle zweier PGAS-Zugriffe, dann ist zusätzlich noch ein expliziter Test auf eine happened-before-Relation zwischen den beiden Ereignissen notwendig.
Mit Hilfe dieser Regeln können die Tests auf data races weiter beschleunigt werden. Für jeden PGAS-Zugriff werden pro Thread nur noch die direkten Speicherzugriffe getestet, die im jeweiligen Zeitintervall des PGAS-Zugriffs liegen. Da die Zeitstempel als einfache aufsteigende Indices aufgefasst werden können, ist das Finden der fraglichen direkten Speicherzugriffe anhand eines Zeitintervalls schnell in konstanter Zeit möglich. Der Test der PGAS-Zugriffe untereinander kann mithilfe von Intervallbäumen [Ber08] erfolgen. Die zweite Dimension ist der Adressraum, denn nur sich sowohl zeitlich als auch im Adressbereich überlappende PGAS-Zugriffe müssen weiter untersucht werden. Die Berechnung der Zeitintervalle kann anhand eines beliebigen Threads

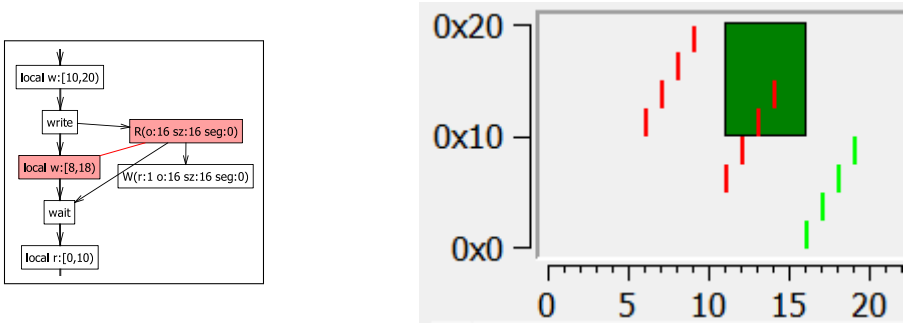


Abbildung 5.12: Ein data race zwischen direkten Speicherzugriffen und einem PGAS-Speicherzugriff

vorgenommen werden. Unter der Annahme, dass der Hauptthread eines Ranks die meisten PGAS-Operationen und -Synchronisationen auf die eigenen Segmente vornimmt, sollte diese Berechnung in Bezug auf diesen Thread geschehen.

In Speicherzugriffsdiagrammen sind data races intuitiv als Überlagerung mehrerer Zugriffe erkennbar. Aus Satz 3 folgt, dass jede überlappende Darstellung eines direkten Speicherzugriffs e und eines PGAS-Speicherzugriffs p in einem Speicherzugriffsdiagramm ein data race ist, wenn mindestens ein Zugriff schreibend ist ($e_a = WRITE \vee p_a = WRITE$). Eine Überlagerung auf der x-Achse (Zeit) bedeutet $e \not\prec p \wedge p \not\prec e$. Eine Überlagerung auf der y-Achse (Adressraum) bedeutet $m_e \cap m_p \neq \emptyset$. Damit sind die Bedingungen von Korollar 1 erfüllt.

Abbildung 5.12 verdeutlicht diesen Sachverhalt anhand des bereits für Abbildung 5.9 verwendeten Listings 8. Für das gezeigte Speicherzugriffsdiagramm wurde der Startindex der Schleife auf Zeile 8 auf 2 geändert. Damit beschreibt diese Schleife auch den Speicherbereich ab 0x10. Durch den vorhergehenden `gaspi_write`-Aufruf wurde jedoch auch eine asynchrone Leseoperation auf diesen Bereich gestartet, die erst durch das `gaspi_wait` auf Zeile 10 beendet wird. Der Wert der Feldelemente an Position 0x10 und 0x14 ist damit im Moment des Lesens nicht determiniert. Im Speicherzugriffsdiagramm ist die Überlagerung der Zugriffe deutlich zu erkennen. Außerdem erfolgt eine Markierung der betroffenen Programmereignisse im Task Graphen. Der Programmierer erhält somit Informationen sowohl über die zu dem data race führenden zeitlichen und räumlichen Überlagerungen als auch über den Kontext der entsprechenden Zugriffsereignisse.

PGAS-Speicherzugriffe werden in Speicherzugriffsdiagrammen zusätzlich gerahmt. Damit sind

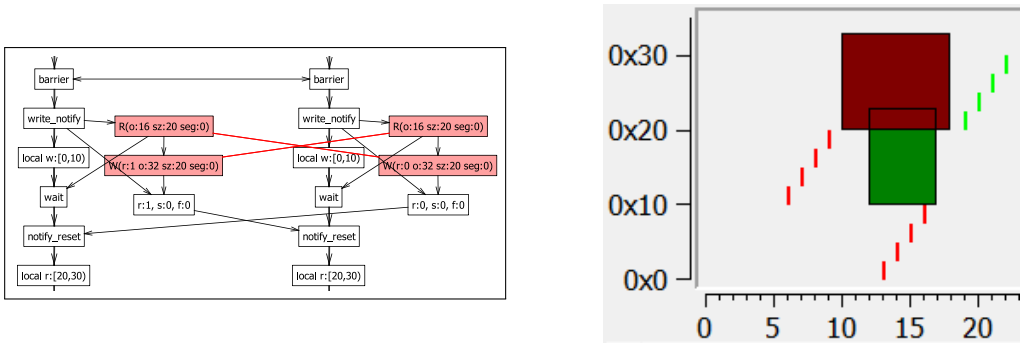


Abbildung 5.13: Data race zwischen PGAS-Speicherzugriffen, im Speicherzugriffsdiagramm von Rank 0 als Überlappung sichtbar

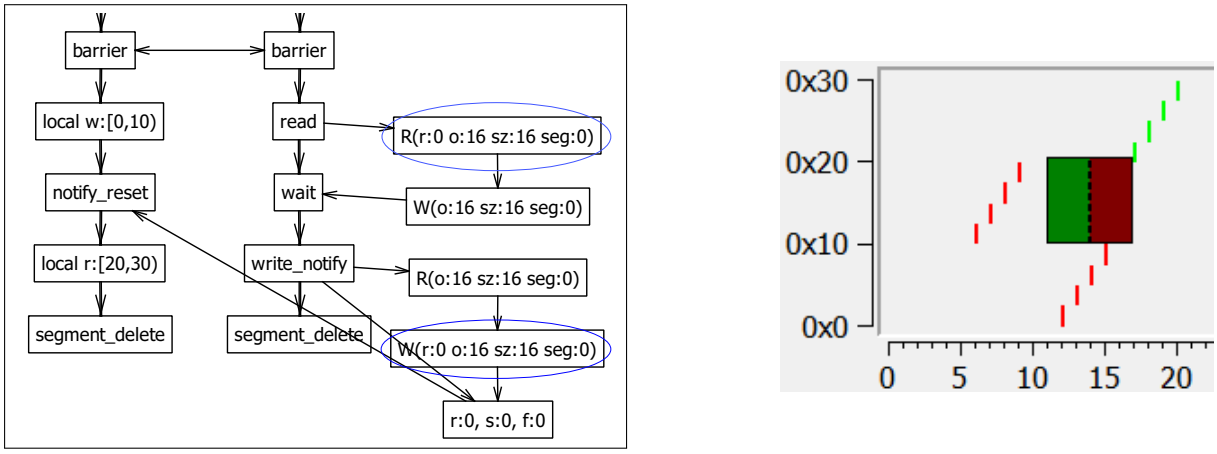


Abbildung 5.14: Darstellung sich nicht überlagernder, entfernter Speicherzugriffe auf Rank 0

eventuelle Überlappungen von Zugriffen erkennbar. Das Programm, mit dem das Speicherzugriffsdiagramm in Abbildung 5.13 erstellt wurde, unterscheidet sich nur dahingehend von dem in Abbildung 5.10 dargestellten Programm, dass die Anzahl der übertragenen Daten vergrößert wurde. Die Offsets der Daten im Segment bleiben gleich. Dadurch reicht der Adressbereich des lokalen asynchronen Lesezugriffs in den Bereich des entfernten Schreibzugriffs. Da beide Zugriffe parallel zueinander ablaufen, kommt es zu einer Überlagerung und somit zu einem data race.
Satz 4 schließt zwar eine parallele Ausführung zweier PGAS-Speicherzugriffe aus, wenn deren Zeitintervall sich nicht überlagert. Daraus kann jedoch nicht geschlossen werden, dass bei einer Überlagerung der Zeitintervalle zwingend auch eine parallele Ausführung stattfindet. Abbildung 5.14 zeigt diesen Sonderfall, der auftritt, wenn die PGAS-Speicherzugriffe mit einer happened-before-Relation verbunden sind, deren Pfad nicht über den betrachteten Thread läuft. Der auf Rank 1 laufende Thread liest zuerst Daten von Rank 0 und schreibt diese später an dieselbe Stelle zurück. Aus Sicht des Threads auf Rank 0 beginnt sowohl das asynchrone Lesen als auch Schreiben der Daten im zugehörigen Segment an der anfänglichen Barriere. Beendet werden beide Zugriffsoperationen dann durch den Erhalt der Notifikation. Die entsprechenden Ereignisse sind im Task Graphen blau markiert. Tatsächlich können der Lese- und Schreibzugriff nicht parallel stattfinden. Aufgrund des zwischengeschalteten `gaspi_wait`-Aufrufs wird der Schreibzugriff erst nach Fertigstellung des Lesezugriffs erfolgen. Eine überlappende Darstellung wäre in diesem Fall irreführend. Stattdessen werden – wie in Abbildung 5.14 gezeigt – alle auf diese Weise miteinander verbundenen PGAS-Speicherzugriffe auf das gesamte Zeitintervall gleichmäßig aufgeteilt und die zeitliche Begrenzung durch eine gestrichelte Linie dargestellt. Diese besondere Darstellung wird jedoch nur gewählt, wenn sich auch die Adressbereiche überlagern. Ansonsten werden die Zugriffe jeweils über das gesamte Zeitintervall dargestellt.

5.3 Interaktive Kombination von Speicherzugriffsdiagrammen und Task-Graph-Visualisierung

Im bisherigen Verlauf des Kapitels wurde deutlich, dass Speicherzugriffsdiagramme und Task-Graph-Visualisierung zum Verständnis eines Programmlaufs auf verschiedene Weise beitragen.

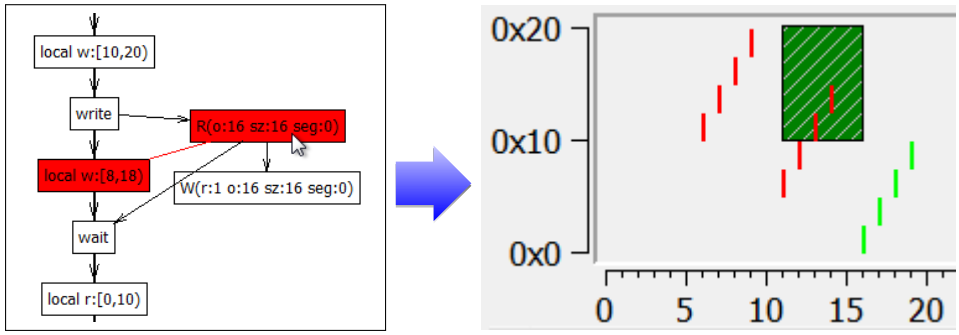


Abbildung 5.15: Auswahl eines asynchronen Speicherzugriffs im Task Graph – der gewählte Speicherzugriff wird im Diagramm schraffiert dargestellt

Eine Task-Graph-Visualisierung gibt vor allem einen Überblick über den gesamten Programmverlauf und zeigt die logischen Zusammenhänge zwischen Prozessen auf. Speicherzugriffsdiagramme dienen der detaillierteren Untersuchung einzelner Threads und deren Speicherzugriffsmuster. Nichtsdestotrotz basieren beide Analyseformen auf dem gleichen, im vorhergehenden Kapitel eingeführten Modell. Damit liegt es nahe, in einem Werkzeug eine interaktive Verbindung der beiden Visualisierungen zu schaffen.
Wählt der Programmanalyst in einem Task Graphen ein Speicherzugriffsereignis aus, so kann dieses direkt im Speicherzugriffsdiagramm dargestellt werden. Direkte und lokale asynchrone Speicherzugriffsereignisse führen dabei immer zum auslösenden Thread. Für entfernte Speicherzugriffe muss in hybriden Programmen auch ein anzuzeigender Thread ausgewählt werden. Bei dieser Auswahl wird zuerst getestet, ob das Speicherzugriffsereignis Teil eines data races ist. In diesem Fall wird der Thread gewählt, der den mit diesem Ereignis kollidierenden Speicherzugriff ausgelöst hat. Ist jedoch auch der kollidierende Speicherzugriff ein entferntes Zugriffsereignis oder ist das Speicherzugriffsereignis nicht Teil eines data races, dann wird der mutmaßliche Hauptthread des entsprechenden Ranks gewählt, welcher `gaspi_proc_init` aufgerufen hat.
Abbildung 5.15 stellt diesen Arbeitsablauf für das Beispiel aus Abbildung 5.12 dar. Der Mauszeiger im Task Graph verdeutlicht die Position des Klicks. Im Speicherzugriffsdiagramm wird das ausgewählte Zugriffsereignis markiert. Bei der Auswahl eines direkten Speicherzugriffs wird ein Begrenzungsrahmen um alle Zugriffe des Ereignisses gezeichnet (Abbildung 5.16). Damit ist es dem Programmanalysten möglich, den Kontext von Speicherzugriffen schnell zu finden.

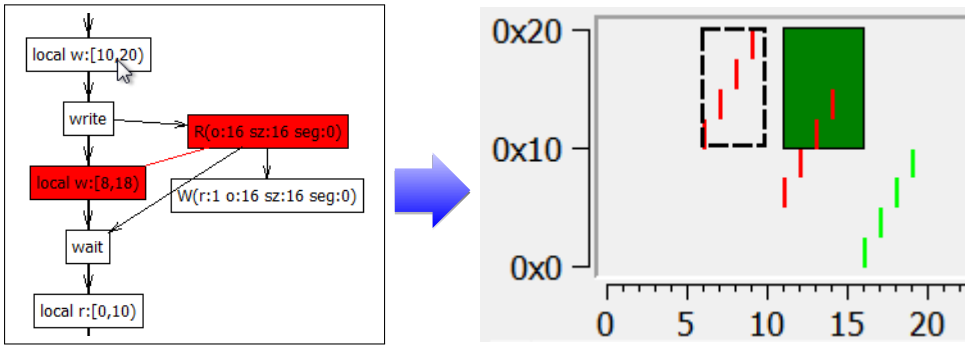


Abbildung 5.16: Auswahl eines direkten Speicherzugriffsereignisses im Task Graph – die enthaltenen Speicherzugriffe werden im Diagramm umrahmt

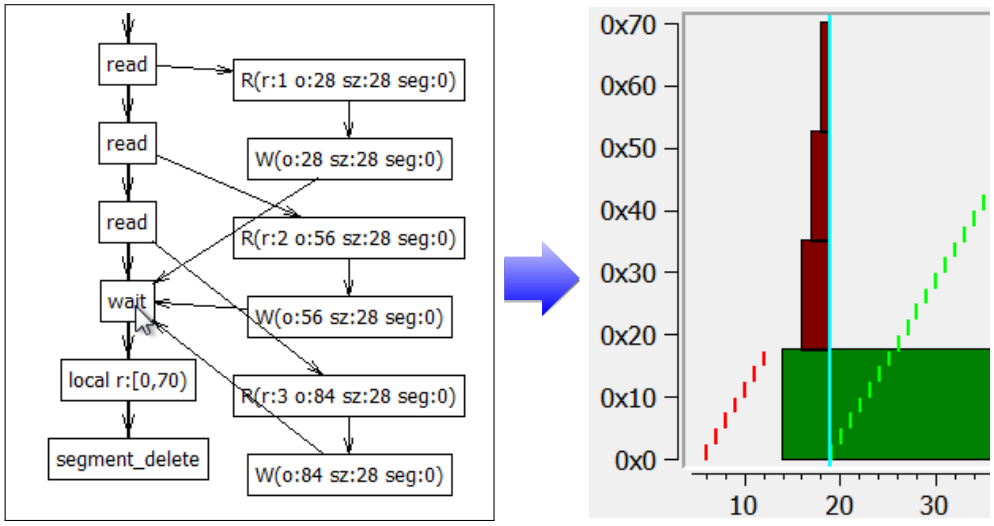


Abbildung 5.17: Auswahl eines Funktionsaufrufs im Task Graph – der Zeitstempel der Funktion wird im Speicherzugriffsdiagramm blau markiert

Wählt der Programmanalyst im Task Graphen einen Funktionsaufruf aus, so wird das Speicherzugriffsdiagramm des auslösenden Threads dargestellt und die Position des Funktionsaufrufs innerhalb der Speicherzugriffssequenz markiert. Damit kann die Wirkung von Funktionen auf die Begrenzung von asynchronen Speicherzugriffen besser verstanden werden. Dieser Ablauf ist in Abbildung 5.17 für die Auswahl einer `gaspi_wait`-Funktion dargestellt. Im zugeordneten Speicherzugriffsdiagramm wird dann anhand der blauen Markierung deutlich, dass die drei durch vorhergehende `gaspi_read`-Aufrufe ausgelösten lokalen asynchronen Schreibzugriffe durch diese Funktion begrenzt werden. Der durch andere Prozesse ausgelöste entfernte Lesezugriff im unteren Adressbereich wird dagegen von der `gaspi_wait`-Funktion nicht begrenzt.
Genauso, wie es möglich ist, von der Task-Graph-Visualisierung zu Speicherzugriffsdiagrammen interaktiv zu wechseln, kann auch ein umgekehrter Wechsel sinnvoll sein. Prinzipbedingt können in Speicherzugriffsdiagrammen nur Speicherzugriffe, nicht jedoch Funktionsaufrufe ausgewählt werden. Bei der Auswahl eines direkten Speicherzugriffs wird im Task Graph das entsprechende Zugriffsereignis markiert. Interessanter ist die Auswahl eines asynchronen Speicherzugriffs. In diesem Fall wird im Task Graphen nicht nur das Zugriffsereignis markiert, sondern es werden auch die Pfade zum Thread des Speicherzugriffsdiagramms markiert, die zur zeitlichen Begrenzung des Speicherzugriffs führen. Damit können asynchrone Speicherzugriffe nicht nur im Kontext anderer Speicherzugriffe, sondern auch im Kontext der auslösenden und synchronisierenden Funktionen analysiert werden.
Für lokale asynchrone Speicherzugriffe wird dazu im Task Graphen die auslösende Funktion und normalerweise das synchronisierende `gaspi_wait` angezeigt. In manchen Fällen (z. B. im Stencil-Code von Listing 6) findet statt einer direkten Synchronisation mit `gaspi_wait` eine implizite Synchronisation über Notifikationen statt. Abbildung 5.18 zeigt diesen Fall. Im Speicherzugriffsdiagramm wird ein lokaler asynchroner Lesezugriff ausgewählt, der Teil der Übertragung der Halo-Daten an einen Nachbar-Thread ist. In der Task-Graph-Darstellung wird dieser Lesezugriff daraufhin grau hinterlegt. Außerdem wird der Pfad zu den Operationen, die diesen Lesezugriff zeitlich begrenzen, blau dargestellt. Der Zugriff beginnt mit Aufruf des auslösen-

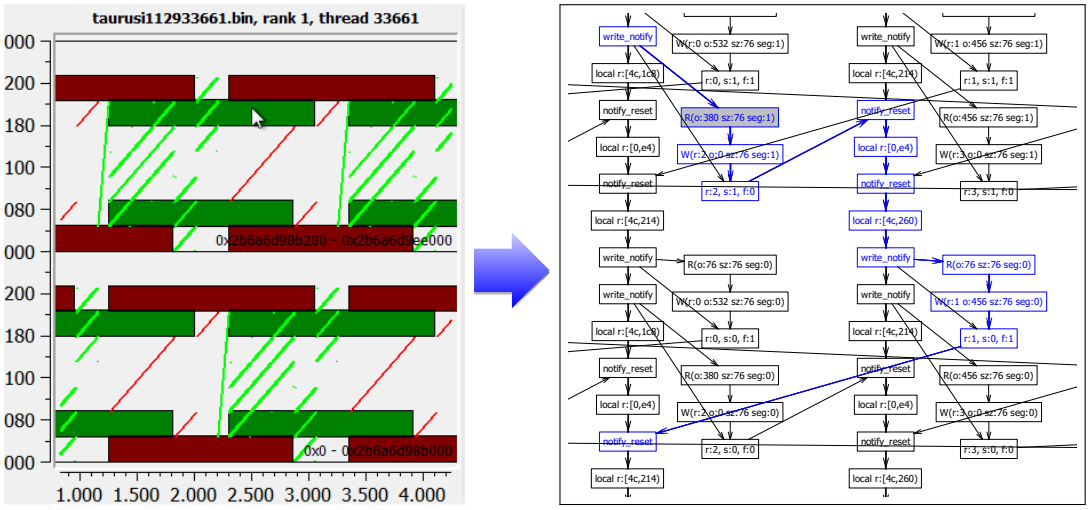


Abbildung 5.18: Auswahl eines lokalen asynchronen Speicherzugriffs im Speicherzugriffsdiagramm – die Pfade zu den begrenzenden Operationen werden im Task Graph blau markiert

den `gaspi_write`. Sicher beendet ist der Zugriff, wenn der Thread eine Rückmeldung über die Berechnung der nächsten Iteration vom Nachbar-Thread erhält. Der entsprechende Synchronisationspfad kann in der Task-Graph-Darstellung gut nachvollzogen werden.
Die Auswahl entfernter asynchroner Speicherzugriffe wird prinzipiell genauso behandelt. Es wird immer der Bezug zum Thread des Speicherzugriffsdiagramms hergestellt. In Abbildung 5.19 ist dieser Fall wiederum für den Stencil-Code dargestellt. Nachdem ein entfernter Schreibzugriff ausgewählt wurde, können in der Task-Graph-Darstellung die kausalen Zusammenhänge zu den Operationen des eigenen Threads erkannt werden. Im dargestellten Fall beginnt der entfernte Schreibzugriff bereits mit der Übertragung der Halo-Daten der vorherigen Iteration. Beendet wird der Schreibzugriff direkt durch eine `gaspi_notify_reset`-Operation.

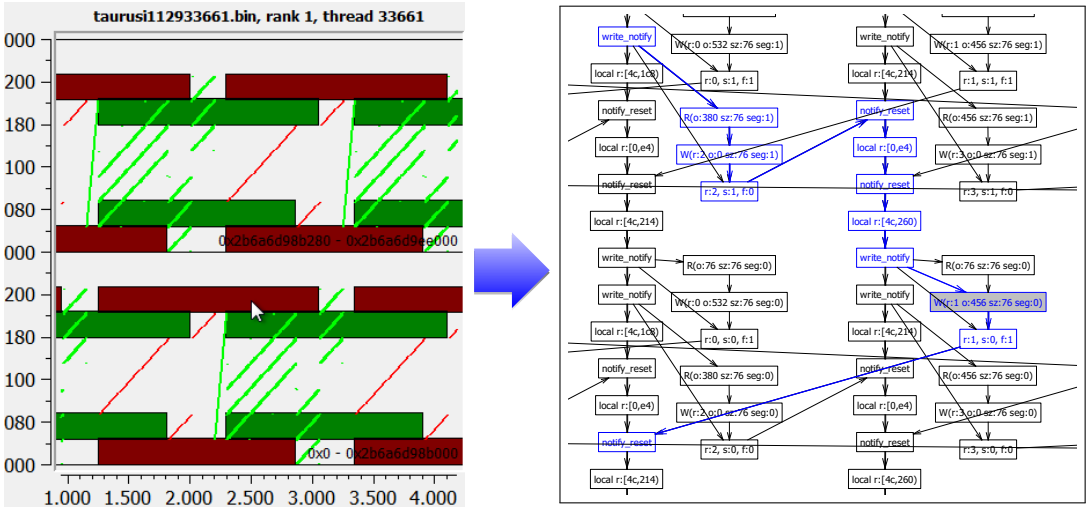


Abbildung 5.19: Auswahl eines entfernten asynchronen Speicherzugriffs im Speicherzugriffsdiagramm – der kausale Zusammenhang zu den Operationen des zum Speicherzugriffsdiagramm gehörenden Threads wird deutlich

5.4 Performance-Analyse und Optimierung von PGAS-Speicherzugriffen

Die Untersuchung von Speicherzugriffen kann auch ein Mittel zur Analyse der Programmeffizienz sein. Bei der Einführung von Speicherzugriffsdiagrammen am Beispiel der Sortierverfahren (Abbildung 5.8) wurde schon die Möglichkeit erwähnt, aus der Anzahl und Abfolge der direkten Speicherzugriffe Schlüsse auf die Effizienz zu ziehen. Tatsächlich existieren vielfältige Methoden, Speicherzugriffe in Bezug auf ihre Effizienz zu analysieren (siehe Abschnitt 3.2.3). Diese Arbeit fokussiert sich auf die Untersuchung von Speicherzugriffen im Kontext von PGAS-Systemen. Deswegen werden im Folgenden Möglichkeiten zur Effizienzanalyse erörtert, die speziell die effiziente Einbettung von PGAS-Operationen in die Programmausführung zum Ziel haben.
Zum einen sollten Kommunikationsoperationen so früh wie möglich gestartet werden, um während des Kommunikationsprozesses möglichst viele weitere Berechnungen vornehmen zu können. Zum anderen sollte eine Kommunikation auch erst dann synchronisiert werden, wenn die Daten benötigt werden. So kann die Entkopplung von Kommunikation und Synchronisation zur Verbesserung der Programmeffizienz beitragen. Speicherzugriffsdiagramme können diese Prinzipien gut veranschaulichen und bei der Suche nach derartigen Verbesserungen sehr hilfreich sein.
Bei der Betrachtung des Adressraums eines PGAS-Speicherzugriffs in einem Speicherzugriffsdiagramm sind auch die vor bzw. nach dem Zugriff stattfindenden Speicherzugriffe ersichtlich. Insbesondere ist erkennbar, ob zeitliche Lücken zwischen dem PGAS-Speicherzugriff und anderen Zugriffen auf diesen Adressraum existieren. Existiert eine solche Lücke vor dem Start eines PGAS-Speicherzugriffs, so kann der Start des Zugriffs eventuell nach vorn verschoben werden. Andersherum kann eine Lücke nach Beendigung eines PGAS-Speicherzugriffs auf eine zu zeitige Synchronisation hindeuten.
Zur Ableitung von Hinweisen auf mögliche Effizienzverbesserungen anhand dieser Lücken in einem Speicherzugriffsdiagramm ist eine systematische Betrachtung der zeitlichen Begrenzungen der PGAS-Speicherzugriffe nützlich. Bezogen auf einen Thread können diese Begrenzungen von PGAS-Speicherzugriffen auf zwei verschieden Arten verursacht werden:

Explizit: der Pfad von der begrenzenden Operation zum Speicherzugriff (bzw. umgekehrt) führt nicht über Tasks eines anderen Threads.

Implizit: der Pfad von der begrenzenden Operation zum Speicherzugriff (bzw. umgekehrt) führt über mindestens einen Task eines anderen Threads.

Ruft ein Thread eine PGAS-Operation auf, so ist der damit erfolgte Start des lokalen PGAS-Speicherzugriffs immer explizit. Mit Aufruf von `gaspi_write` startet der lokale asynchrone Lesezugriff und mit Aufruf von `gaspi_read` startet entsprechend der lokale asynchrone Schreibzugriff. Das Beenden eines lokalen PGAS-Speicherzugriffs kann explizit durch Aufruf von `gaspi_wait` erfolgen. Für durch `gaspi_read` ausgelöste lokale Schreibzugriffe gibt es nur diese Möglichkeit. Die von `gaspi_write` ausgelösten Lesezugriffe können jedoch auch implizit beendet werden, wenn sich der sendende Prozess später wieder mit dem empfangenden Prozess synchronisiert. Im Task Graphen von Abbildung 5.18 ist diese implizite Beendigung eines Lesezugriffs anhand des

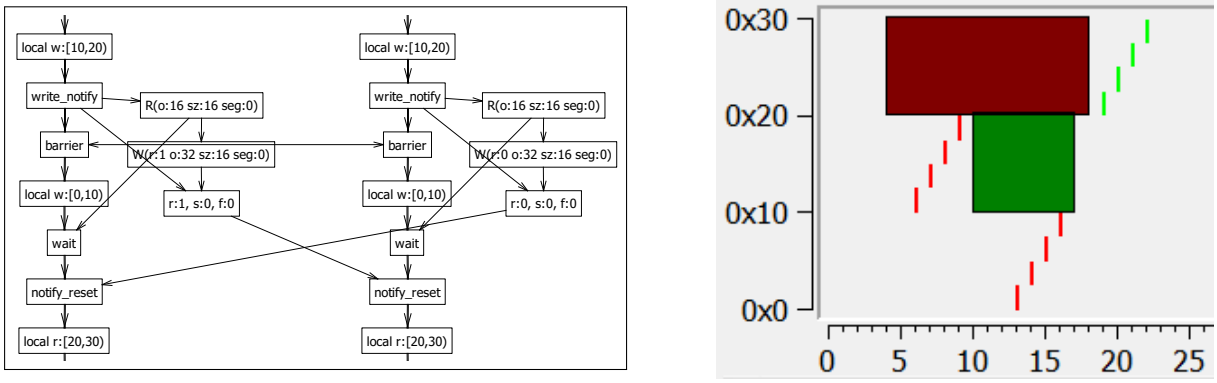


Abbildung 5.20: Optimierung der Aufrufposition einer PGAS-Operation und die Auswirkung auf das Speicherzugriffsdiagramm

blau markierten Synchronisationspfades vom ausgewählten Lesezugriff zum auslösenden Thread gut zu erkennen.

Der Start eines entfernten PGAS-Speicherzugriffs erfolgt aus Sicht des betrachteten Prozesses immer implizit. Diese für einseitige Kommunikationssysteme zentrale Eigenschaft bedeutet, dass sich Prozesse an einem beliebigen Punkt vor einem Zugriff zur Vermeidung von Zugriffskonflikten synchronisieren müssen. Beendet werden von `gaspi_read` ausgelöste entfernte lesende PGAS-Speicherzugriffe immer implizit. Dazu muss der auslösende Thread `gaspi_wait` aufrufen und sich dann mit dem Zielprozess synchronisieren. Dagegen müssen entfernte schreibende PGAS-Speicherzugriffe explizit beendet werden, indem auf den Erhalt einer Notifikation getestet wird.

Bei der Untersuchung eines Speicherzugriffsdiagramms zur Optimierung von PGAS-Speicherzugriffen kann der Programmierer die Unterscheidung zwischen expliziten und impliziten Begrenzungen verwenden, um Optimierungsstrategien und Optimierungsaufwand abschätzen zu können. Generell sind explizite Begrenzungen leichter zu optimieren, da die beteiligten Operationen direkt identifizierbar sind. In dem Fall helfen bereits oft bessere Aufrufpositionen für diese Operationen. Das Bewegen einer impliziten Begrenzung bedarf dagegen einer tiefgreifenderen Analyse der Kommunikationsstruktur. Eine Optimierung einer solchen Grenze ist demzufolge auch oft mit einer algorithmischen Änderung verbunden.

Als Ausgangspunkt für ein Beispiel der Optimierung einer expliziten Begrenzung dient das Programm, dessen Speicherzugriffsdiagramm in Abbildung 5.10 dargestellt ist. In diesem Diagramm ist im Adressintervall von 0x10 bis 0x20 eine zeitliche Lücke zwischen dem letzten direkten Schreibzugriff und dem Start des asynchronen Lesezugriffs zu erkennen. Im Task Graph ist zu erkennen, dass diese Lücke durch den zwischenzeitlichen Aufruf der Barriere entsteht. Der Start des asynchronen Lesezugriffs wird durch die Position von `gaspi_write` explizit festgelegt. Entsprechend kann die zeitliche Lücke geschlossen werden, indem der Aufruf der Funktion vor die Barriere verschoben wird. Das aus dieser Änderung resultierende Speicherzugriffsdiagramm und der zugehörige Task Graph sind in Abbildung 5.20 dargestellt. Der asynchrone Lesezugriff startet jetzt direkt nach den direkten Schreibzugriffen, wodurch die Datenübertragung und die Ausführung der Barriere überlagert stattfinden können.

In Abbildung 5.20 kann auch erkannt werden, dass sich durch die Programmänderung auch der implizite Start des PGAS-Schreibzugriffs verschoben hat. Da im vorliegenden Beispiel vor

diesem Schreibzugriff keine direkten Zugriffe auf den entsprechenden Adressraum stattgefunden haben, führt die Optimierung der Aufrufposition von `gaspi_write` nicht zu einem data race. In komplexeren Programmen sind die Konsequenzen solcher Optimierungen jedoch oft schwer abzuschätzen. In [HKK17] wird daher das automatische Finden einer sowohl optimierten als auch korrekten Aufrufposition für eine gegebene PGAS-Operation beschrieben. Neue Aufrufpositionen werden dem Programmierer als Vorschläge präsentiert, die dieser dann endgültig verifizieren und das Programm entsprechend ändern muss.
Eine automatisierte Performance-Analyse nach dem hier beschriebenen Modell auf einer kompletten Programmausführung würde die Menge der zu untersuchenden PGAS-Operationen sowohl für das Analyseprogramm als auch für den Programmierer abhängig von der Menge der Prozesse machen. Für hoch skalierende PGAS-Programme ist ein solches Vorgehen nicht praktikabel. Daher wurde ebenfalls in [HKK17] eine weitere Anwendung eingeführt, die mit dem hier eingeführten Task-Graph-Modell realisiert wurde. Diese Anwendung stellt eine Performance-Analyse unter Verwendung der Methode des kritischen Pfades dar [YM88]. Die Optimierung von PGAS-Speicherzugriffen kann dann entlang des kritischen Pfades erfolgen, wodurch der Optimierungsaufwand unabhängig von der Skalierung des Programmlaufs wird.
Zur Berechnung des kritischen Pfades ist die Ermittlung von Wartezuständen in einem Task Graphen notwendig. Asynchrone einseitige Kommunikations- und Synchronisationsereignisse schaffen dabei eine Reihe eigenständiger Herausforderungen. Insbesondere beim Vorliegen asynchroner Ordnungsrelationen wie sie z. B. beim Zusammenspiel von `gaspi_write` und `gaspi_notify` auftreten (siehe auch Tabelle 4.3), ist die Ermittlung des kritischen Pfades anspruchsvoll. Für diese Fragestellungen wurden in [HKK17] Lösungen erarbeitet. Da sich die vorliegende Arbeit auf Speicherzugriffsanalyse fokussiert, wird auf eine detaillierte Beschreibung dieser Lösungen jedoch verzichtet.

6 Prototypische Realisierung und Evaluierung

Truth is rarely pure and never simple. [Wil90]

Die Evaluierung der in dieser Arbeit eingeführten Methoden zur Programmanalyse wird anhand einer prototypischen Implementierung vorgenommen. Nach einer Erläuterung der technischen Details dieser Implementierung wird in diesem Kapitel eine qualitative und quantitative Bewertung durchgeführt. Die qualitative Evaluierung des Modells und der darauf aufbauenden Anwendungskonzepte wird anhand von praktischen GASPI-Anwendungen vorgenommen. Dieser Teil demonstriert, welche Analyseergebnisse sich mit Hilfe der entwickelten Werkzeuge und Verfahren erzielen lassen. Im nachfolgenden Teil werden quantitative Untersuchungen zur Skalierbarkeit der Algorithmen sowohl anhand von GASPI-Anwendungen als auch anhand von Testprogrammen durchgeführt.
Alle Messungen wurden auf Haswell-Knoten des Taurus-Clusters am ZIH der TU Dresden durchgeführt. Auf diesen Knoten sind Intel® Xeon® E5-2680 v3 Prozessoren mit 2,50GHz Taktfrequenz im Einsatz. Das installierte Betriebssystem ist ein Red Hat Enterprise Linux Server. Die Zeiten der untersuchten Algorithmen in den Abschnitten 6.3.2 und 6.3.3 wurden mit `clock_gettime(CLOCK_THREAD_CPUTIME_ID, ...)` gemessen.

6.1 Prototypische Realisierung eines Werkzeugs zur Speicherzugriffsanalyse

Das im Rahmen dieser Arbeit entwickelte Werkzeug besteht aus zwei Komponenten. *Memaccessrecord* zeichnet den Programmlauf auf. *Memaccessviewer* wertet die Aufzeichnung post-mortem aus und visualisiert die Analyseergebnisse in den in Kapitel 5 beschriebenen Formen.

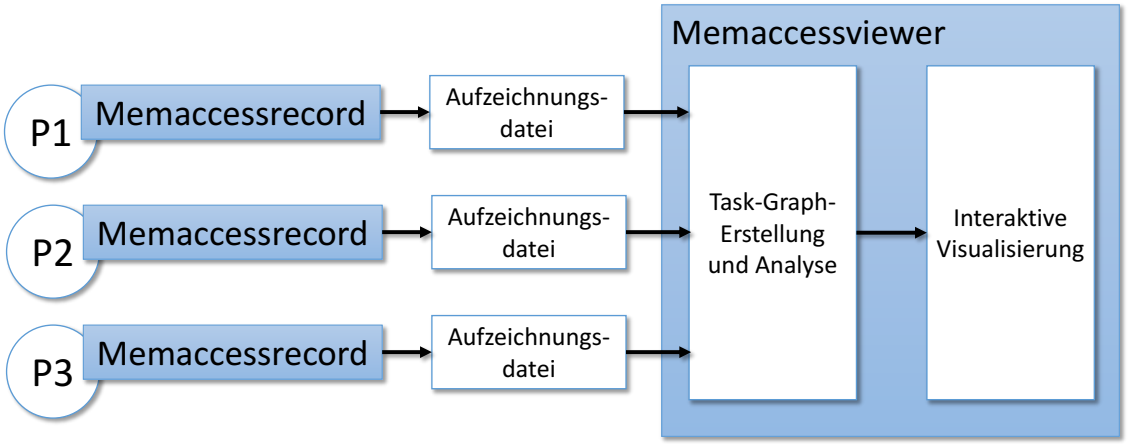


Abbildung 6.1: Zusammenspiel der einzelnen Komponenten des Analysewerkzeuges

Abbildung 6.1 zeigt den Arbeitsablauf bei der Analyse eines GASPI-Programms bestehend aus den drei Prozessen P1, P2 und P3. Beim Start des Programms wird jeder GASPI-Prozess von Memaccessrecord instrumentiert. Während der Abarbeitung eines Prozesses werden die erfassten Programmereignisse in einer Aufzeichnungsdatei gespeichert. Damit entsteht für jeden Prozess eine separate Datei, in der die Ereignisse aller Threads des jeweiligen Prozesses gespeichert sind. Nach Abschluss des Programmlaufs liest Memaccessviewer alle Aufzeichnungsdateien ein und generiert den Task Graphen. Der Nutzer kann dann interaktiv sowohl die Speicherzugriffsdiagramme der einzelnen Threads als auch den Task Graphen über alle Threads des GASPI-Programms untersuchen.

6.1.1 Die Aufzeichnung von direkten Speicherzugriffen

Das Werkzeug Memaccessrecord zur Aufzeichnung von Programmereignissen und direkten Speicherzugriffen basiert auf dem dynamischen Instrumentierungs-API PIN [LCM+05]. PIN kann ein Programm zur Laufzeit auf Instruktionsebene um weiteren Code erweitern. Damit ist eine sehr feingranulare Instrumentierung und Aufzeichnung möglich, die für die hoch aufgelöste Erfassung von direkten Speicherzugriffen notwendig ist. Das für Memaccessrecord wichtigste vom PIN-API bereitgestellte Konzept ist ein *TRACE*. In der PIN-Nomenklatur ist ein TRACE ein Programmabschnitt, der am Zielpunkt einer Verzweigung beginnt und bis zum nächsten unbedingten Sprung reicht, welcher auch ein `call` oder `return` sein kann. Im Gegensatz zu einem Basic Block [Pat11] werden bei der Ausführung eines TRACEs also nicht zwingend alle Instruktionen durchlaufen, da bedingte Sprünge aus dem TRACE heraus erlaubt sind.
Memaccessrecord setzt voraus, dass TRACEs während der Programmausführung konstant bleiben, es unterstützt also keinen selbst-modifizierenden Code. Damit sind auch die Eigenschaften der in einem TRACE enthaltenen Instruktionen und deren Abfolge konstant. Für jede Instruktion kann statisch festgestellt werden, ob sie lesend oder schreibend auf den Speicher zugreift sowie die Anzahl der von einem Zugriff betroffenen Bytes. Spezielle Instruktionen mit variabler Byteanzahl (z. B. solche mit `rep`-Präfixen) werden von PIN zur Instrumentierungszeit in Schleifen umgebaut. Dadurch ist es während der Aufzeichnung ausreichend, zu jedem ausgeführtem TRACE die Identifikationsnummer des TRACEs und daran anschließend die Adressen der Speicherzugriffe zu speichern. Wird ein TRACE vorzeitig verlassen, werden die restlichen Speicherzugriffe auf 0 gesetzt. Tatsächliche Speicherzugriffe auf die Adresse 0 werden also nicht unterstützt. Abbildung 6.2 stellt das Speicherformat von Memaccessrecord schematisch dar. Die im statischen Teil gezeigten Daten werden nur einmal pro Programmaufzeichnung gespeichert. Dadurch wird die mit der Dauer des Programms mitwachsende Datenmenge auf die wesentlichen Informationen reduziert.
Neben direkten Speicherzugriffen zeichnet Memaccessrecord auch die Programmereignisse *Betreten* und *Verlassen* einer Funktion auf. Zusätzlich können Argumente und Rückgabewerte der Funktion aufgezeichnet werden. Das beinhaltet auch Rückgabewerte per Referenz, deren Werte während der Aufzeichnung durch Dereferenzierung des entsprechenden Funktionsargumentes erfasst werden. Die Aufzeichnung von Funktionen und deren Argumenten ist konfigurierbar.
Für die in dieser Arbeit gezeigten Anwendungsmöglichkeiten wurden ausgewählte Funktionen des GASPI-APIs, des pthreads-APIs und des OpenMP-APIs aufgezeichnet. Die Namen

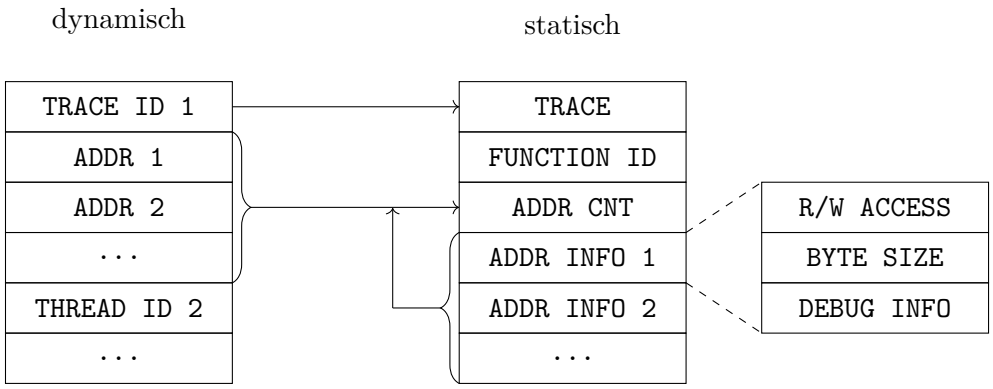


Abbildung 6.2: Struktur des Dateiformats zur Aufzeichnung der Speicherzugriffe. Im statischen Teil werden für jedes TRACE alle konstanten Informationen gespeichert. Im dynamischen Teil werden entsprechend der Programmausführung fortlaufend die TRACEs und nachfolgend die Adressen der Zugriffe gespeichert. Während der Analyse werden aus dieser Sequenz und den zugehörigen statischen Adress-Informationen der TRACEs die Speicherzugriffe rekonstruiert.

von OpenMP-Funktionen sind nicht standardisiert und können für unterschiedliche Compiler-Konfigurationen voneinander abweichen. In diesen Fällen muss die Konfiguration entsprechend angepasst werden. Alle Speicherzugriffe, die während der Ausführung einer erfassten Funktion auftreten, werden nicht mit aufgezeichnet. Insbesondere werden so direkte Speicherzugriffe in GASPI-Segmente durch GASPI-Funktionen ausgefiltert. Bisher nicht implementiert ist die damit geschaffene Möglichkeit, die in Funktionen wie z. B. `memcpy` auftretenden Speicherzugriffe durch die Aufzeichnung eines einzelnen Funktionsaufrufs zu ersetzen.
Memaccessrecord implementiert mehrere ziel-spezifisch einsetzbare Methoden, um die durch die direkten Speicherzugriffe entstehende Datenmenge weiter zu reduzieren. Zugriffe über die Register `esp` und `ebp` werden nicht instrumentiert. Für solche Zugriffe wird angenommen, dass die Zieladresse im Bereich des Stacks liegt, der nicht Teil eines GASPI-Segments sein kann. Weiterhin werden bereits während der Aufzeichnung alle TRACEs von der Speicherung ausgenommen, die keinen Zugriff in ein GASPI-Segment enthalten. Die zu speichernden Daten werden während der Aufzeichnung in Blöcke aufgeteilt und vor dem Abspeichern mit LZ4 [Col11] gepackt. LZ4 wurde gewählt, da er bei guten Komprimierungsraten sehr schnell arbeitet und außerdem eine einfache Integration in Memaccessrecord möglich war.

6.1.2 Visualisierung der Programmaufzeichnung

In Memaccessviewer sind zwei Komponenten zusammengefasst. Die erste Komponente liest die von Memaccessrecord erstellten Aufzeichnungsdateien ein, erstellt daraus mit den in Kapitel 4 beschriebenen Verfahren den Task Graphen und führt über diesen Graphen eine Data-Race-Analyse durch. Die Ergebnisse dieser Schritte können als Textausgabe an den Nutzer gemeldet werden. Optional wird der Task Graph der zweiten Komponente übergeben, welche die Visualisierung des Graphen und die Darstellung der Speicherzugriffsdiagramme übernimmt.
Die Visualisierungskomponente muss mehrere Anforderungen erfüllen. Da aufgrund der Datenmenge nicht alle Speicherzugriffsereignisse ständig im Speicher vorgehalten werden können, müs-

sen die jeweils darzustellenden Ereignisse auf Anforderung aus der Aufzeichnungsdatei gelesen werden. Des Weiteren muss eine Transformation des Adressraums auf die y-Achse unter der Beachtung möglicher Faltungen (siehe Abschnitt 5.2 (Seite 65)) möglich sein. Außerdem ist für den Wechsel zwischen Speicherzugriffsdiagrammen und Task-Graph-Ansicht die detaillierte Auswertung von Nutzerinteraktionen notwendig. Vorgefertigte Software zur Diagrammdarstellung wie z. B. gnuplot [WKm13] oder SciDAVis [sci17] konnte jeweils nicht alle Anforderungen erfüllen. Aus diesen Gründen wurde eine eigene Entwicklung aufbauend auf der Qwt-Bibliothek [Rat16] implementiert. Qwt basiert auf der plattformübergreifenden Bibliothek Qt [qt118] und stellt Komponenten zur Darstellung z. B. von Legenden oder Achsen in Diagrammen bereit.
Die Ermittlung von relevanten Prozessparametern wie der Rank oder die Adressen von GASPI-Segmenten geschieht durch die Auswertung aufgezeichneter Funktionen wie `gaspi_proc_rank` und `gaspi_segment_ptr`. Diese Funktionen müssen also von jedem Prozess des untersuchten GASPI-Programms aufgerufen werden. Prozesse, die diese Funktionen nicht aufrufen, sind jedoch wenig sinnhaft, da solche Prozesse ihre Stellung innerhalb des Prozessraums nicht kennen bzw. nicht lokal auf das eigene GASPI-Segment zugreifen können.

6.2 Analyse von Anwendungen aus der Forschungspraxis

In diesem Abschnitt werden die eingeführten Analysemethoden anhand der in Tabelle 6.1 genannten GASPI-Programme evaluiert. Die Entwicklung dieser Programme geschah unabhängig von dieser Arbeit. Sie können deswegen auch als Gradmesser für die Einsatzfähigkeit der entwickelten Methoden und Werkzeuge dienen. Des Weiteren werden anhand der Beispiele auch mögliche Erweiterungen des Task-Graph-Modells erläutert.
Das Testprogramm *alltoall* ist ein Beispielprogramm, welches u. a. im Rahmen von GASPI-Tutorials verwendet wird. Das Programm *stencil* wurde bereits in Kapitel 5 vorgestellt. Es wurde am ITWM Kaiserslautern entwickelt. Am ZIH der TU Dresden erfuhr dieses Programm im Rahmen von wissenschaftlichen Arbeiten [Her15, HKK17] und unter Zuhilfenahme der hier vorgestellten Werkzeuge eine gründliche Analyse und Weiterentwicklung.
Die Anwendungen *async3d* und *cfdproxy* sind Teil der PGAS-community-benchmarks [Sim14b]. Diese Programm-Kollektion kann für Vergleiche verschiedener Parallelisierungs- und Kommunikationsstrategien verwendet werden. Meist werden mehrere MPI- und GASPI-Varianten zur Verfügung gestellt. In dieser Arbeit wird jeweils die hybrid parallelisierte GASPI-Variante der Berechnungskerne von *async3d* und *cfdproxy* untersucht.

Kürzel	Beschreibung
alltoall	Globale Datenverteilung
stencil	Stencil-Code mit Double Buffering
async3d	Dreidimensionale Stencilberechnung mit inneren Datenabhängigkeiten
cfdproxy	CFD-Berechnung über unstrukturierte Gitter

Tabelle 6.1: Evaluierte GASPI-Programme

6.2.1 Korrektheitsanalyse einer Routine zur Datenverteilung

Der GASPI-Standard umfasst neben der API-Spezifikation auch eine Reihe von Beispielprogrammen, mit denen Programmierprinzipien in einem PGAS-System demonstriert werden. Eines dieser Programme implementiert `alltoall`-Funktionsvarianten, mit der jeder Prozess Datenelemente zu jedem anderen Prozess verteilt. Listing 9 zeigt die schematische Implementierung einer frühen Version dieser Funktion. Zuerst berechnet jeder Prozess lokal ein Element des Feldes, wobei die Position des Elementes innerhalb des Segmentes vom Rank des Prozesses bestimmt wird. Nachdem ein Prozess seine Berechnung abgeschlossen hat, ruft er die Funktion `alltoall` auf. Diese Funktion schreibt zuerst das lokal berechnete Element, welches sich an Position `own_data_offset` im Segment `s` befindet, in die Segmente aller anderen Prozesse (Zeilen 5,6). Die Position des Elementes bleibt dabei auf den entfernten Segmenten die gleiche wie im lokalen Segment. Danach wartet die Funktion zuerst lokal auf die Fertigstellung der asynchronen `gaspi_write`-Aufrufe (Zeile 11) und danach auf alle anderen Prozesse (Zeile 12). Nach der Barriere sollten die Berechnungsergebnisse der anderen Prozesse im lokalen Segment vorliegen. Die Funktion `verfiy_local_segment_data` (Zeile 14) verifiziert diese Tatsache.

```
1  function alltoall
2  {
3    own_data_offset = own_rank * element_size;
4
5    for_all (p : each other process)
6      gaspi_write(s, own_data_offset,
7                  p, s, own_data_offset,
8                  element_size,
9                  queue);
10
11   gaspi_wait(queue);
12   gaspi_barrier(GASPI_GROUP_ALL);
13
14   verfiy_local_segment_data();
15 }
```

Listing 9: Fehlerhafte GASPI-Implementierung einer alltoall-Prozedur

Diese Implementierung ist tatsächlich einige Zeit auf einem Infiniband-Cluster korrekt gelaufen. Beim Einsatz auf einem Ethernet-Cluster kam es jedoch öfters zu Fehlschlägen der Funktion `verfiy_local_segment_data`. Bei der ursprünglichen Programmierung ging man irrtümlicherweise davon aus, dass nach Verlassen der `gaspi_wait`-Funktion nicht nur die lokalen Lesezugriffe, sondern auch die entfernten Schreibzugriffe der `gaspi_write`-Funktion beendet sind. Durch die nachfolgende Barriere wäre dann sichergestellt, dass auch die Schreibzugriffe anderer Prozesse auf das lokale Segment beendet sind. In der Tat garantiert das Verlassen von `gaspi_wait` jedoch nur die Komplettierung der lokalen Speicherzugriffe.

Durch den Einsatz der in dieser Arbeit beschriebenen Analysemethoden konnte der Fehler schnell aufgedeckt werden. In Abbildung 6.3 ist der Task Graph eines alltoall-Laufs von vier Prozessen dargestellt. Die Adressen aller Zugriffe wurden auf den Start der Segmente normiert. Jeder Prozess berechnet zuerst ein 28 Byte großes Datenfeld und schreibt dieses in das lokale Segment

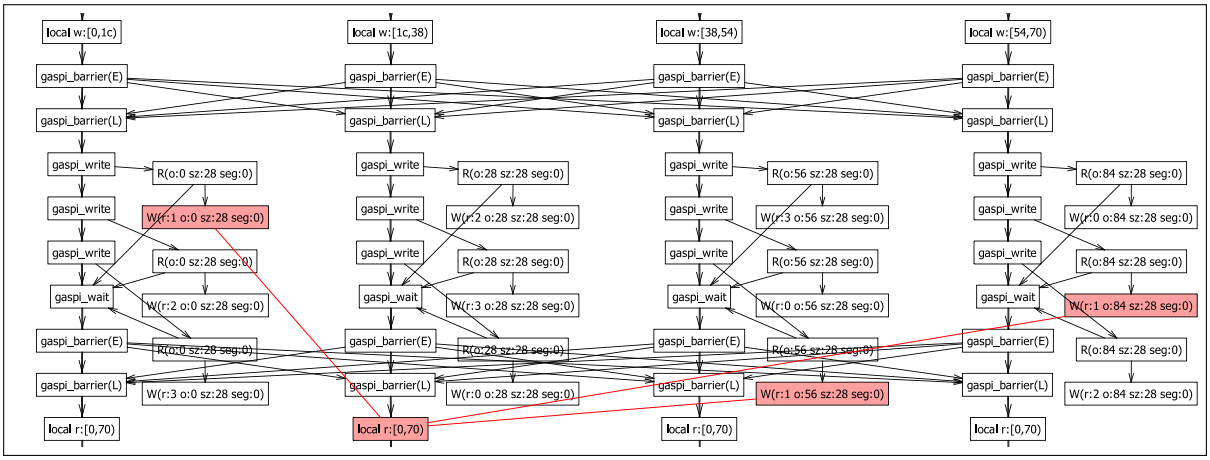


Abbildung 6.3: Task Graph der fehlerhaften alltoall-Prozedur

(`local w:x,x+1c`). Nach einer kollektiven Warteoperation wird die alltoall-Prozedur aufgerufen. In dieser überträgt der Prozess die berechneten Daten an alle anderen Prozesse.

Die asynchronen entfernten Schreibereignisse von Rank 1 sind rot markiert. Diese stehen in Konflikt mit dem ebenfalls markierten lokalen Leseereignis nach dem `gaspi_barrier`-Aufruf. Die markierten und durch rote Linien miteinander verbundenen Ereignisse formen data races. Jedes markierte asynchrone Schreibereignis greift auf 28 Byte große Teile (`sz:28`) von Segment 0 (`seg:0`) auf Rank 1 (`r:1`) zu. Das markierte, lokale Leseereignis repräsentiert die Leseoperationen über das gesamte lokale Datenfeld von Rank 1. Diese werden von der Verifizierungsfunktion durchgeführt. Die markierten Ereignisse greifen also auf gleiche Speicherbereiche zu, stehen jedoch in keiner $\prec$-Relation zueinander. Tatsächlich kann man erkennen, dass die Fertigstellung der rot markierten, entfernten Schreibereignisse an keiner Stelle synchronisiert wird, da auf diese Ereignisse keine weiteren Ereignisse folgen.

Diese Tatsache trifft auch auf die nicht markierten Schreibereignisse zu anderen Zielprozessen zu. Auch diese Ereignisse verursachen data races mit den jeweiligen lokalen Lesezugriffen der Verifizierungsfunktion. Auf eine Markierung dieser Ereignisse wurde in dieser Abbildung jedoch zu Demonstrationszwecken verzichtet. Aber auch für die markierten Ereignisse ist nicht sofort ersichtlich, welche Bereiche genau die data races verursachen und wie die asynchronen Schreibzugriffe im Kontext der lokalen Speicherzugriffe eingeordnet sind. Task Graphen eignen sich also nur bedingt zur Untersuchung von data races.

In Abbildung 6.4 ist deshalb das Speicherzugriffsdiagramm desselben Programmlaufs auf Rank 1 dargestellt. Zuerst erfolgt die Berechnung des lokalen Elementes, wobei nacheinander 4 Byte große Daten in das im Segment befindliche Feld geschrieben werden. Erkennbar ist diese Berechnung an den bis Zeitschritt 14 gehenden, aufsteigenden Schreibzugriffen von Offset 0x1c (28) bis 0x38 (56). Danach beginnt die Verteilung der Daten auf die anderen Ranks mittels `gaspi_write`. Die dazugehörigen dunkelgrünen Blöcke im Speicherzugriffsdiagramm repräsentieren die asynchronen lokalen Lesezugriffe. Lokal synchronisiert und damit begrenzt werden diese Zugriffe durch den Aufruf von `gaspi_wait`. Währenddessen schreiben die anderen Prozesse ihre Ergebnisse ebenfalls mittels `gaspi_write` in das Segment. Wie man im Diagramm gut erkennen kann, betreffen diese von den dunkelroten Blöcken repräsentierten Schreibzugriffe alle

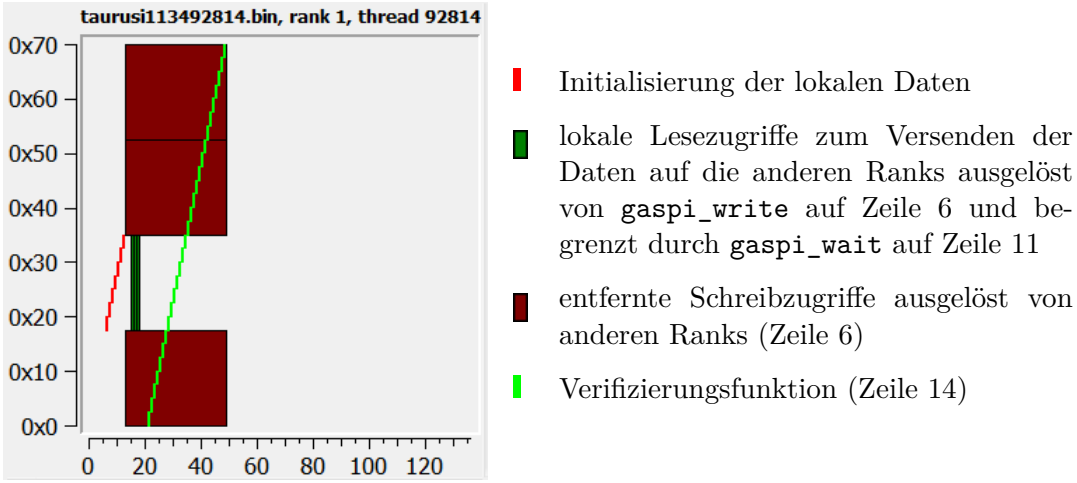


Abbildung 6.4: Speicherzugriffsdiagramm der fehlerhaften alltoall-Prozedur auf Rank 1 (Zeilenangaben beziehen sich auf Listing 9)

nicht lokal berechneten Elemente. Die lokalen Lesezugriffe der Verifizierungsfunktion sind dann beginnend ab Zeitschritt 20 als aufsteigende Lesezugriffe über das gesamte Datenfeld sichtbar. Die Überlappung der direkten Lesezugriffe und der von anderen Prozessen verursachten asynchronen Schreibzugriffe ist in der Abbildung deutlich erkennbar. Wie in Abschnitt 5.2.2 ausgeführt, repräsentiert diese Überlappung ein data race. Ebenfalls gut erkennbar ist der Kontext, in dem das data race auftritt und auch, welche Korrekturen notwendig sind. Im vorliegenden Fall sind die asynchronen Schreibzugriffe vor Aufruf der Verifizierungsfunktion noch nicht beendet. Diese müssen also auf dem Zielprozess synchronisiert werden. Das muss prinzipiell durch die Nutzung von `gaspi_notify` geschehen, denn nur Notifikationen können durch `gaspi_write` ausgelöste Schreibzugriffe synchronisieren. Für eine solche Lösung benötigt jeder Prozess für jeden anderen Prozess eine dedizierte Notifikation, was allerdings zu einer eingeschränkten Skalierbarkeit führt.

Eine andere Lösung des Problems ist in Listing 10 dargestellt. Diese Implementierung wartet am Eingang der Funktion auf die Fertigstellung der Berechnungen aller anderen Prozesse, liest dann

```
1  function alltoall
2  {
3    gaspi_barrier(GASPI_GROUP_ALL);
4
5    own_data_offset = own_rank * element_size;
6    for_all (p : each other process)
7      gaspi_read(s, own_data_offset,
8                 p, s, own_data_offset,
9                 element_size,
10                queue);
11
12   gaspi_wait(queue);
13   verfiy_local_segment_data();
14 }
```

Listing 10: Korrigierte GASPI-Implementierung der alltoall-Prozedur

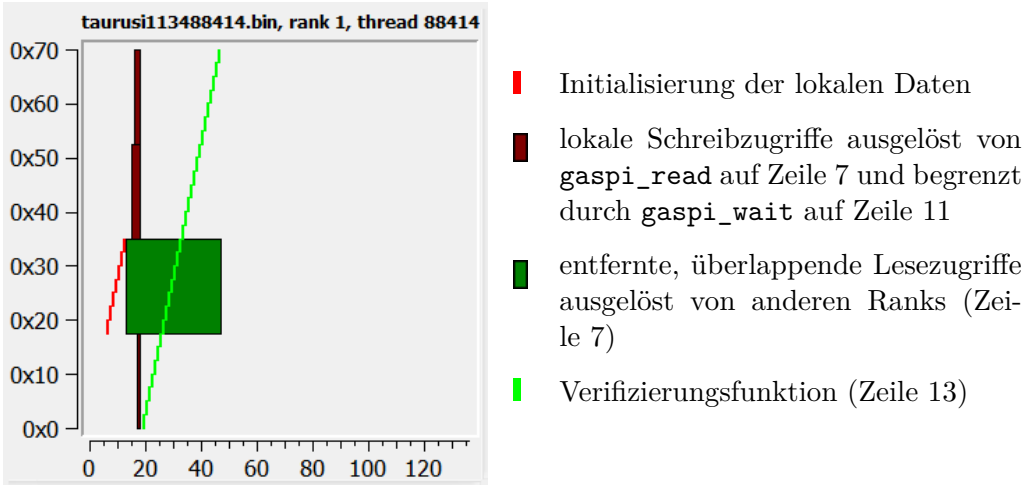


Abbildung 6.5: Speicherzugriffsdiagramm der korrigierten alltoall-Prozedur auf Rank 1 (Zeilenangaben beziehen sich auf Listing 10)

die Daten von diesen Prozessen und schreibt sie in das eigene Segment. Die Daten werden jetzt also entfernt gelesen und lokal geschrieben. Anders als in der ursprünglichen Lösung werden durch den Aufruf von `gaspi_wait` sowohl der entfernte Lesezugriff als auch der lokale Schreibzugriff mit dem lokalen Prozess synchronisiert. Die Synchronisation des entfernten Lesezugriffs auf dem jeweiligen entfernten Prozess könnte dann z. B. mittels einer Barriere erfolgen. Wie aus dem Speicherzugriffsdiagramm in Abbildung 6.5 hervorgeht, ist das im vorliegenden Fall jedoch gar nicht notwendig, da nun die entfernten Zugriffe Lesezugriffe sind. Diese überlappen sich zwar mit den lokalen Lesezugriffen, bilden aber keine data races mehr.

6.2.2 Optimierung eines zweidimensionalen Stencil-Codes

Die in Listing 6 (Seite 62) bereits gezeigte GASPI-Implementierung eines eindimensionalen Stencil-Codes wird in GASPI-Tutorials als eine mögliche Lösung dieser Klasse von Programmierproblemen verwendet. Diese Lösung demonstriert das Erreichen einer sehr hohen Effizienz und Skalierbarkeit durch Ausnutzung der Möglichkeiten asynchroner einseitiger Kommunikationssysteme. In [HKK17] konnte jedoch anhand der in Abschnitt 5.4 beschriebenen Methode gezeigt werden, dass sich die Effizienz dieses Stencil-Codes sogar noch weiter erhöhen lässt.
Abbildung 6.6 zeigt einen Ausschnitt aus dem Speicherzugriffsdiagramm eines Threads des originalen Stencil-Codes. Der angezeigte Adressraum ist in der Mitte durch die durchgezogene schwarze Linie gefaltet. Der untere Teil zeigt Segment 0, der obere Teil Segment 1. In jeder Iteration werden Daten aus einem Segment gelesen und Ergebnisse in das andere Segment geschrieben, was gut an den direkten Speicherzugriffen zu erkennen ist. In der bei knapp vor dem Zeitstempel 400 beginnenden ersten Iteration am Anfang des Diagramms wird aus Segment 1 gelesen und in Segment 0 geschrieben. Zuerst berechnet `compute_inner_part` den inneren Teil des Datenfeldes. Kurz nach Zeitstempel 500 beginnt die Berechnung der Randbereiche (`compute_below_part`, `compute_above_part`). Diese Berechnung dauert bis zum Zeitstempel 700. Damit ist eine Iteration vollständig und Quell- und Zielsegmente werden vertauscht. In der Iteration ab Zeitstempel 700 wird dann aus Segment 0 gelesen und in Segment 1 geschrieben.

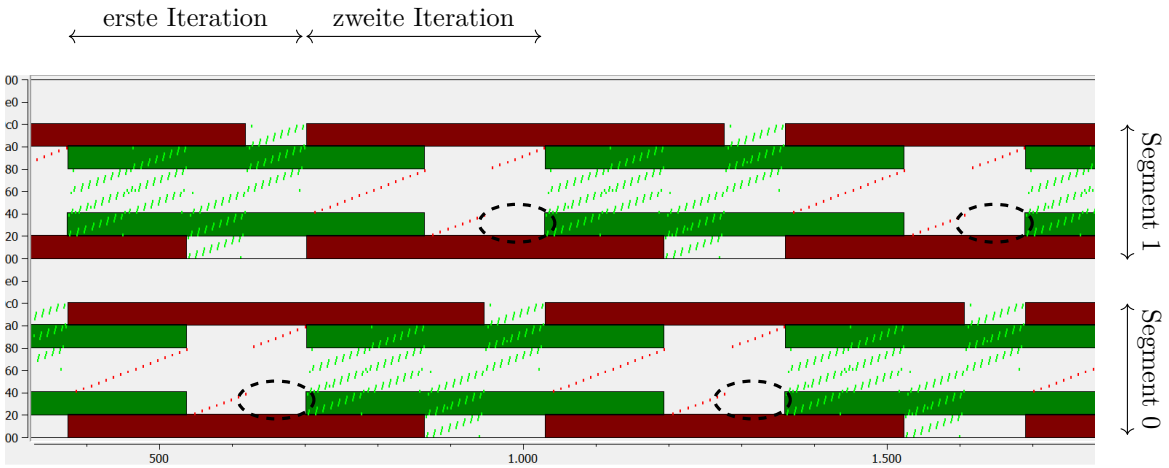


Abbildung 6.6: Speicherzugriffsdiagramm des Stencil-Codes von Listing 6. Die Ellipsen markieren systematische zeitliche Lücken zwischen direkten Schreibzugriffen und dem Start asynchroner Lesezugriffe auf demselben Speicherbereich.

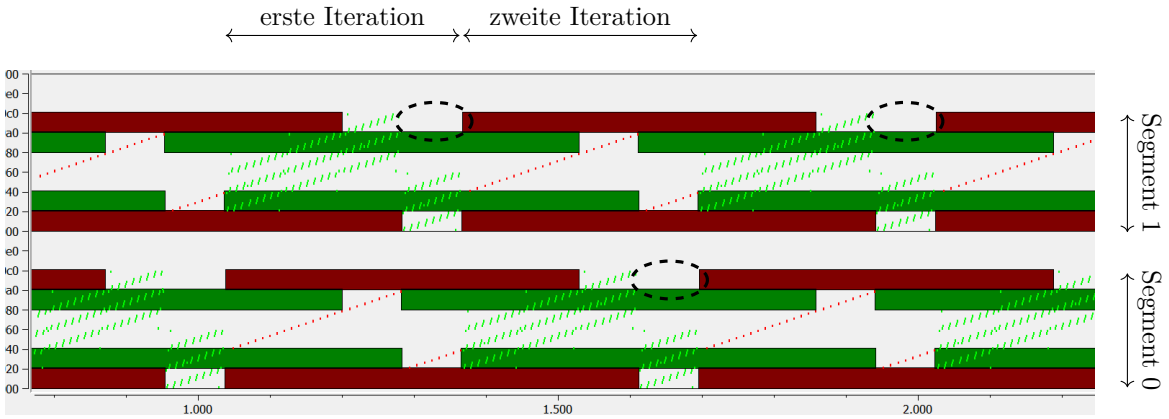


Abbildung 6.7: Speicherzugriffsdiagramm des Stencil-Codes von Listing 11. Die in Abbildung 6.6 markierten Lücken sind verschwunden. Dafür sind an den markierten Stellen systematisch Lücken zwischen Lesezugriffen und dem Start entfernter Schreibzugriffe entstanden.

Wie im Speicherzugriffsdiagramm gut zu erkennen ist, überlagern sich die PGAS-Speicherzugriffe zum Austausch der Halos und die Berechnungen der einzelnen Teile. Die Lesezugriffe werden immer von den lokalen `gaspi_write_notify`-Aufrufen auf den Zeilen 8-14 ausgelöst. Die Schreibzugriffe werden von denselben Aufrufen – allerdings von entfernter Seite aus – ausgelöst. Die lokalen asynchronen Leseoperationen weisen systematische Lücken zu direkten Schreiboperationen in denselben Adressraum auf. Insbesondere sind vor dem expliziten Start der Leseoperationen immer wieder die in der Abbildung markierten Lücken zu vorherigen direkten Schreiboperationen zu erkennen. Das betrifft in beiden Segmenten den Adressraum 0x20-0x40. Der Grund für diese Lücke besteht darin, dass erst beide Ränder des lokalen Datenfeldes berechnet werden, bevor die Ergebnisse zu Beginn der nächsten Iteration an die Nachbarn gesendet werden. Tatsächlich kann aber jeder Rand schon direkt nach erfolgter Berechnung gesendet werden.

In Listing 11 ist diese Optimierung dargestellt. Das Aufrufen von `gaspi_write` geschieht nun getrennt von `gaspi_notify`. Dadurch wird nur der Start der PGAS-Speicherzugriffe geändert, die

```
1  function stencil
2  {
3    for (i = 0; i < max; ++i)
4    {
5      gaspi_notify(own_rank + 1, ...);
6      gaspi_notify(own_rank - 1, ...);
7
8      src_seg = i % 2;
9      target_seg = 1 - src_seg;
10     compute_inner_part(src_seg, target_seg);
11
12     for (k = 0; k < 2; ++k)
13       recv_id = gaspi_wait_reset(seg_id, notify_id, 2);
14
15       if (recv_id == notify_id)
16         compute_below_part(src_seg, target_seg);
17         gaspi_write(target_seg, ...);
18       else
19         compute_above_part(src_seg, target_seg);
20         gaspi_write(target_seg, ...);
21   }
22 }
```

Listing 11: Optimierung des Starts der Datenübertragung im Stencil-Code

Synchronisationsbeziehungen bleiben gleich. Die Zielsegmente der `gaspi_write`-Aufrufe wurden angepasst, da das Senden der Daten jetzt bereits in der jeweils vorhergehenden Iteration stattfindet. Abbildung 6.7 zeigt das Speicherzugriffsdiagramm dieser Optimierung. Im Vergleich zu Abbildung 6.6 ist zu erkennen, dass die Lücken zwischen den lokalen Berechnungen der Ränder und den PGAS-Lesezugriffen auf die Adressräume 0x20-0x40 und 0x80-0xa0 vollständig verschwunden sind.

Da die Synchronisationsbeziehungen nicht verändert wurden, sollte das optimierte Programm weiterhin korrekt sein. Bei der Evaluierung dieser Optimierung wurde jedoch ein data race festgestellt, welches nur bei Läufen mit zwei Prozessen auftrat. Der Grund für dieses data race liegt in den unterschiedlichen Garantien, die der GASPI-Standard für einen `gaspi_write_notify`-Aufruf und für eine Kombination aus Aufrufen von `gaspi_write` und `gaspi_notify` gibt. Wie in Tabelle 4.3 (Seite 52) gezeigt wird, synchronisiert ein `gaspi_notify`-Aufruf alle vorhergehenden durch `gaspi_write` ausgelösten Schreibzugriffe auf dieselbe Queue und denselben Ziel-Rank. Eine Ordnungsgarantie für nacheinander ausgeführte `gaspi_notify`-Aufrufe existiert nicht. Abbildung 6.8 zeigt einen Ausschnitt des Task Graphen, mit dessen Hilfe das aufgetretene data race schnell verstanden werden konnte. Auf Rank 0 werden durch `gaspi_write` zwei PGAS-Schreibzugriffe ausgelöst. Die Ziele dieser Schreibzugriffe sind der linke und rechte Nachbar des jeweiligen Prozesses. Bei zwei Prozessen sind diese Nachbarn jedoch ein und derselbe Prozess. Damit synchronisieren sich beide PGAS-Schreibzugriffe schon mit dem ersten `gaspi_notify`-Aufruf von Rank 0 (`r:1, s:1, f:1`). Der direkt nachfolgende `gaspi_notify`-Aufruf (`r:1, s:1, f:0`) synchronisiert dadurch keinen PGAS-Schreibzugriff. Die Notifikation dieses Aufrufs wird jedoch zuerst empfangen und direkt mit der Berechnung des entsprechenden Randbereichs begonnen. Unter den gegebenen Umständen garantiert der GASPI-Standard nicht, dass die dafür not-

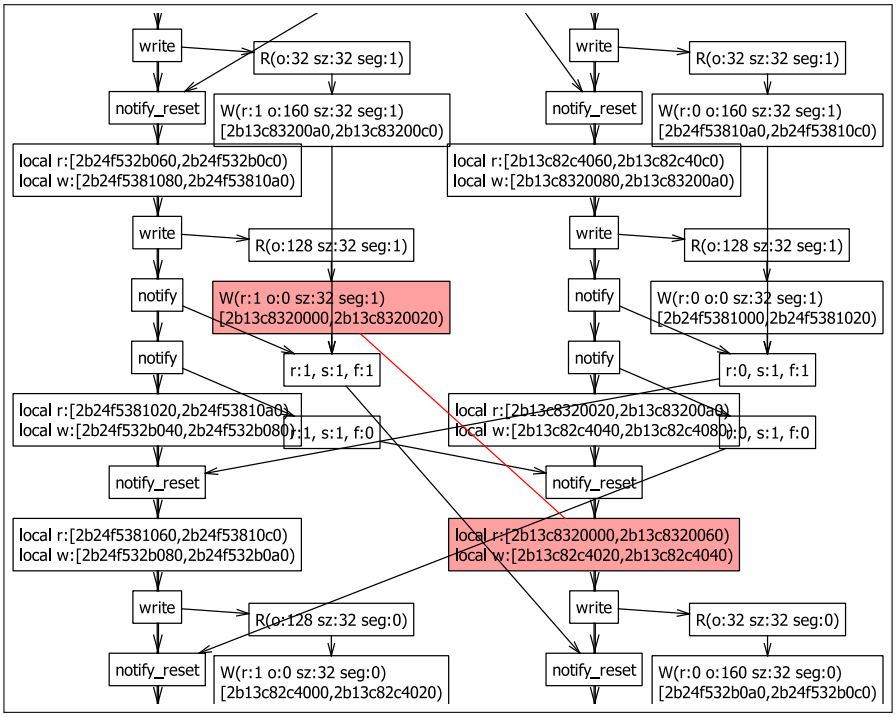


Abbildung 6.8: Data race im Stencil-Code auf genau 2 Ranks

wendigen Daten schon vorliegen. Die Behebung des Fehlers erfolgte, indem für die linken und rechten Nachbarn unterschiedliche Queues verwendet wurden. Dadurch wird die Zuordnung der `gaspi_write` und `gaspi_notify`-Aufrufe wieder eindeutig.

In dem zuvor beschriebenen Optimierungsschritt wurden lediglich die expliziten Begrenzungen der PGAS-Lesezugriffe verändert. Damit konnten jedoch noch nicht die in den Speicherzugriffsdiagrammen sichtbaren systematisch auftretenden Lücken zwischen direkten Lesezugriffen und dem Start von PGAS-Schreibzugriffen in den äußeren Halo-Bereichen geschlossen werden. Die in Abbildung 5.19 (Seite 79) gezeigte interaktive Untersuchung eines PGAS-Schreibzugriffs führt zu der Erkenntnis, dass dessen Start implizit von der Position eines `gaspi_notify`-Aufrufs im

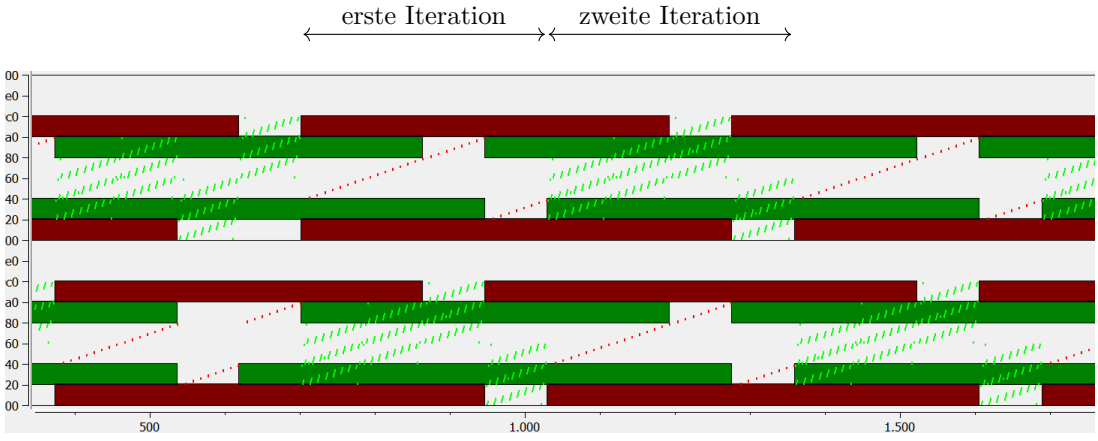


Abbildung 6.9: Speicherzugriffsdiagramm des optimierten Stencil-Codes von Listing 12. Alle systematischen Lücken zwischen direkten Speicherzugriffen und PGAS-Zugriffen sind verschwunden.

```
1  function stencil
2  {
3    for (i = 0; i < max; ++i)
4    {
5      src_seg = i % 2;
6      target_seg = 1 - src_seg;
7      compute_inner_part(src_seg, target_seg);
8
9      for (k = 0; k < 2; ++k)
10       recv_id = gaspi_wait_reset(seg_id, notify_id, 2);
11
12       if (recv_id == notify_id)
13         compute_below_part(src_seg, target_seg);
14         gaspi_write_notify(target_seg, ...);
15       else
16         compute_above_part(src_seg, target_seg);
17         gaspi_write_notify(target_seg, ...);
18   }
19 }
```

Listing 12: Optimierung der Datenübertragung und Synchronisation im Stencil-Code

sendenden Prozess abhängt. Die Optimierung dieser Aufrufposition kann erfolgen, indem auch die `gaspi_notify`-Aufrufe direkt hinter die bereits verschobenen `gaspi_write`-Aufrufe bewegt werden. Damit können die beiden Aufrufe wieder zu einem `gaspi_write_notify`-Aufruf zusammengefasst werden, wodurch sich das in Listing 12 gezeigte Programm ergibt.
Abbildung 6.9 zeigt das Speicherzugriffsdiagramm dieses zweiten Optimierungsschrittes. Im Vergleich zu den vorhergehenden Speicherzugriffsdiagrammen sind nun fast alle Lücken in den Halo-Bereichen eliminiert. Außerdem kann die schrittweise Zunahme der Asynchronität gut nachvollzogen werden. In Abbildung 6.6 starten sowohl die asynchronen Lese- als auch Schreibzugriffe immer mit Beginn einer Iteration. In Abbildung 6.7 ist nur noch der Start aller Schreibzugriffe an die Iterationsgrenzen gebunden. In der vollständig optimierten Variante in Abbildung 6.9 erfolgt auch der Start der Schreibzugriffe nicht mehr synchron zu den Iterationen. Mit dieser vollständigen Asynchronität kann auch das weiterhin sporadisches Auftreten von Lücken erklärt werden, da sich der Berechnungsfortschritt der beiden Halo-Bereiche eines Prozesses so weit voneinander entfernen kann, dass der Prozess auf einen Nachbarn warten muss.
Die beschriebenen Optimierungsschritte wurden auf Taurus evaluiert. Abbildung 6.10 zeigt die Laufzeit und parallele Effizienz der drei beschriebenen Code-Varianten für verschiedene Prozesszahlen. Die Variante *original* bezieht sich auf Listing 6, *write-opt* zeigt die durch die in Listing 11 vorgenommene Verschiebung der `gaspi_write`-Aufrufe erreichbaren Verbesserungen und *write-notify-opt* die Performance von Listing 12, in dem auch die Synchronisation verbessert wurde. Aus den Diagrammen geht hervor, dass die Optimierungen nur für eine große Anzahl von Prozessen eine messbare Wirkung zeigen. Dieses Ergebnis ist nicht überraschend, da die Optimierungsstrategie letztendlich auf eine verbesserte Überlappung von Berechnung und Kommunikation und einen höheren Grad von Asynchronität abzielt. Die Bedeutung dieser Faktoren steigt mit der Anzahl der Prozesse und Threads. Obwohl bereits die originale Implementierung einen sehr hohen Anteil an Überlappung zwischen Kommunikation und Berechnung aufweist,

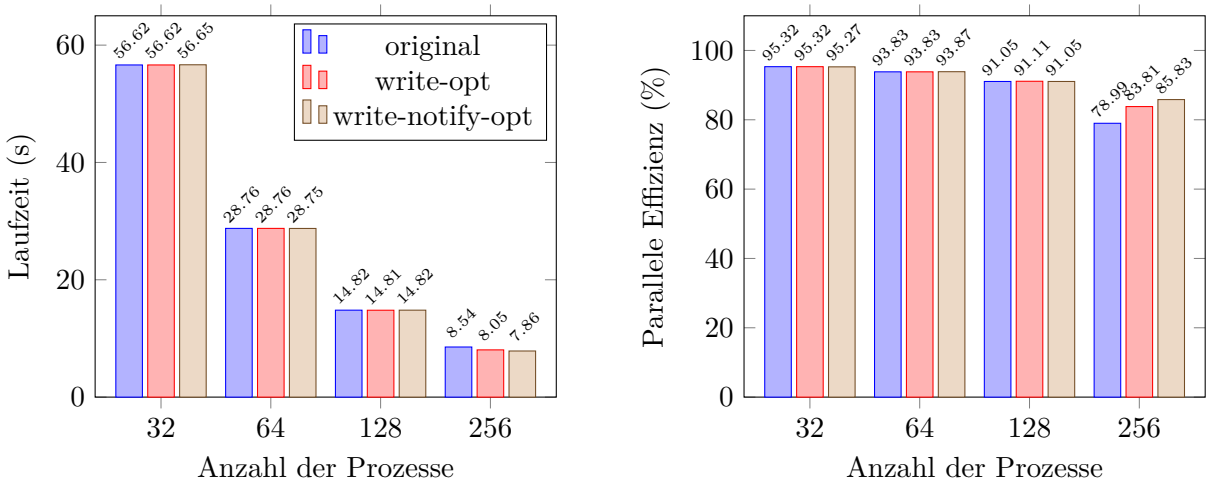


Abbildung 6.10: Vergleich der Laufzeiten und parallelen Effizienz der Stencil-Optimierungen (nach [HKK17])

konnten die beschriebenen Optimierungen die Laufzeit des Programms auf 256 Prozessen nochmals um 8% beschleunigen. Diese Optimierungen wurde dabei lediglich anhand der Analyse der durch das eingeführte Modell aufgezeigten zeitlich-logischen Beziehungen der Speicherzugriffe gefunden. Insbesondere war keine Echtzeitmessung des Programms notwendig, um diese Verbesserungsmöglichkeiten erkennen zu können.

6.2.3 Dreidimensionale Stencil-Berechnung mit inneren Abhängigkeiten

Der *Synch_p2p*-Benchmark ist Teil der *Parallel Research Kernels* [WM14] und implementiert einen Stencil-Code mit inneren Datenabhängigkeiten. Die Anwendung *async3d* baut auf diesem Kernel auf und berechnet einen dreidimensionalen Stencil nach folgender Formel:

```
array_elem(i,j,k) = (array_elem(i-1, j, k) +
                     array_elem(i, j-1, k) +
                     array_elem(i, j, k-1)) * one_third + 1
```

Untersucht wurde die hybrid parallelisierte GASPI/OpenMP-Implementierung [Sim15]. Diese synchronisiert auf GASPI-Ebene mit `gaspi_notify` und `gaspi_notify_reset`. Auf OpenMP-Ebene werden sowohl Barrieren als auch manuell programmierte Fortschrittszähler als Synchronisationsmittel verwendet.

Die GASPI-Synchronisationen und OpenMP-Barrieren werden direkt aufgezeichnet. Für eine Erfassung aller Synchronisationsbeziehungen müssen jedoch auch die Fortschrittszähler ausgewertet werden. Die Implementierung dieses Synchronisationsprinzips ist in Listing 13 gezeigt. Jedem Thread ist ein Zähler zugeordnet. Die Zähler sind in einem Datenfeld organisiert, die Indizierung geschieht über die Thread-ID `tid`. Der ausführende Thread wartet in der Schleife, bis das Feldelement `stage[tid]` einen bestimmten Wert überschreitet. Das Inkrementieren von `stage[tid]` geschieht in einem anderen Thread, wobei jeder Thread nur genau einen Fortschrittszähler inkrementiert. Ein Dekrementieren findet nicht statt. Sowohl das Inkrementieren als auch das Warten sind nicht in separaten Funktionen implementiert, sondern direkt im fortlaufenden Code geschrieben. Das erschwert eine Aufzeichnung dieser Synchronisationsereignisse.

```
1    ...
2    volatile int* it = &stage[tid];
3    while (*it <= required_stage)
4    {
5      _mm_pause();
6    }
7    ...
```

Listing 13: Synchronisation über Fortschrittszähler auf Thread-Ebene

Um eine Programmausführung der vorliegenden Implementierung in das Task-Graph-Modell übertragen zu können, mussten zwei Fragen beantwortet werden:

1. Wie kann die Synchronisation über Fortschrittszähler auf das *POST*/*WAITCLEAR* -Modell, welches nur Flags mit den zwei Zuständen *gesetzt* und *gelöscht* unterstützt, übertragen werden?

2. Wie kann das Inkrementieren der Zähler bzw. das Warten auf diese als eigenständige Programmereignisse erfasst werden?

Eine direkte Abbildung von Fortschrittszählern auf das *POST*/*WAITCLEAR*-Modell ist nicht möglich. Jedoch können sie auf das *POST*/*WAIT*-Modell übertragen werden. Dazu nimmt man an, dass jeder Wert eines Zählers ein eigenes Flag darstellt. Sobald der Wert erreicht oder überschritten wird, gilt dieses Flag als *gesetzt*. Da die Zähler nie dekrementiert werden, bleibt ein einmal gesetztes Flag in diesem Zustand. Eine *CLEAR*-Operation findet also nicht statt. Die Warteschleife auf einen bestimmten Wert repräsentiert eine *WAIT*-Operation auf das entsprechende Flag. Hierbei handelt es sich nicht um eine *WAITCLEAR*-Operation, da der Zähler nicht verändert wird. Es handelt sich damit lediglich um eine Interaktion von *POST* und *WAIT*. Außerdem kann während der Programmausführung auf jedes Flag nur genau eine *POST*-Operation stattfinden. Diese Operation kann eindeutig einem Programmereignis zugeordnet werden, da im vorliegenden Anwendungsfall das Inkrementieren eines Fortschrittszählers einem Thread statisch zugeordnet ist. Da also weder *CLEAR*-Operationen existieren noch konkurrierende *POST*-Operationen auftreten können, führt diese Synchronisationsmethode nicht zu Synchronisations-Races und der Task Graph einer Programmausführung bleibt auch beim Vorliegen von Fortschrittszählern deterministisch. Die Auswertung konnte deswegen als Erweiterung in das Replay-Verfahren integriert werden.
Die Erfassung der Operationen über Fortschrittszähler muss der soeben erläuterten Modell-Anpassung gerecht werden. Insbesondere ist während der Erfassung einer Operation nicht nur die Identität des Zählers selbst, sondern auch der Wert des Zählers wichtig, anhand dessen das Flag bestimmt wird. Lagert man das Inkrementieren und die Warteschleife in zwei separate Funktionen aus, so muss bei der Erfassung der Funktionsaufrufe während der Aufzeichnung nicht nur die Identität des Zählers, sondern auch dessen Wert bestimmt werden. Um die Komplexität des Aufzeichnungs-Werkzeugs überschaubar zu halten, wurde stattdessen eine manuelle Instrumentierung vorgenommen. Listing 14 zeigt diese Instrumentierung sowohl für das Inkrementieren als auch die Warteschleife. Die beiden Funktionen `H_POST` und `H_WAIT` führen keine

Operationen aus, die Funktionsaufrufe können aber bei der Aufzeichnung als Programmereignisse erfasst werden. Beide Funktionen erwarten ein 64-Bit-Ganzzahlargument, welches das Flag kodiert. Dazu wird der aktuelle Wert des Zählers mit der Identität des Zählers innerhalb des Prozesses (der Thread-ID `tid`) und dem Prozessrank kombiniert. Auf diese Weise konnten alle Synchronisationsbeziehungen, die *async3d* verwendet, im Task-Graph-Modell abgebildet werden.

```
1    ++stage[tid];
2    H_POST((stage[tid] << 16) + (tid << 8) + rank);
```

```
1    volatile int* it = stage[tid];
2    while (*it <= required_stage)
3    {
4      _mm_pause();
5    }
6    H_WAIT(((required_stage + 1) << 16) + (tid << 8) + rank);
```

Listing 14: Erfassung der Fortschrittszähler über manuelle Instrumentierung

Abbildung 6.11 zeigt einen Ausschnitt aus dem Task Graph einer Programmausführung mit 4 Prozessen zu je 6 Threads. Es werden nur die Synchronisationsbeziehungen der 6 Threads eines Prozesses gezeigt, auf eine Darstellung der asynchronen Speicherzugriffe wurde verzichtet. Erkennbar sind die kaskadierenden Abhängigkeiten der Threads aufgrund der `H_POST/H_WAIT`-Synchronisationen. Das im Rahmen der Arbeit realisierte Darstellungsprogramm richtet die vertikale Position der Tasks nur an Barrieren aus. Dadurch entsteht in der gezeigten Abbildung der Eindruck, dass die Ausführung der Tasks nach `gomp_barrier` innerhalb des Prozesses balanciert ist. Tatsächlich ist die Arbeit jedoch ungleichmäßig verteilt, da jeder Thread außer dem ersten auf seinen Vorgänger warten muss. Das ist an den rückwärts gerichteten Pfeilen von `H_POST` zu `H_WAIT` zu erkennen.
Abbildung 6.12 zeigt die Speicherzugriffsdiagramme von Thread 0 und Thread 2 eines Prozesses. Dargestellt wird eine Iteration, wobei die Zeitstempel auf der x-Achse lediglich den Ausführungszeitpunkt der Programmereignisse innerhalb des jeweiligen Threads wiedergeben und deswegen nicht miteinander vergleichbar sind. Die Interaktionen zwischen Berechnung und Synchronisation in den markierten Speicherbereichen sind jedoch gut erkennbar.

(A) In diesem Speicherbereich führt der dargestellte Thread die Berechnungen durch. Dazu muss auf die Daten desselben Threads des vorhergehenden Prozesses gewartet werden. Deswegen synchronisiert dieser Thread den entsprechenden asynchronen Schreibzugriff vor dem Beginn der direkten Lesezugriffe. Außerdem muss auf vorhergehende Threads des eigenen Prozesses gewartet werden. Damit kommt es zu einer impliziten Synchronisierung mit asynchronen Schreibzugriffen im Speicherbereich C. Unmittelbar nach der Berechnung startet mit einem asynchronen Lesezugriff auf das letzte Element das Versenden des Resultats an den nachfolgenden Prozess.

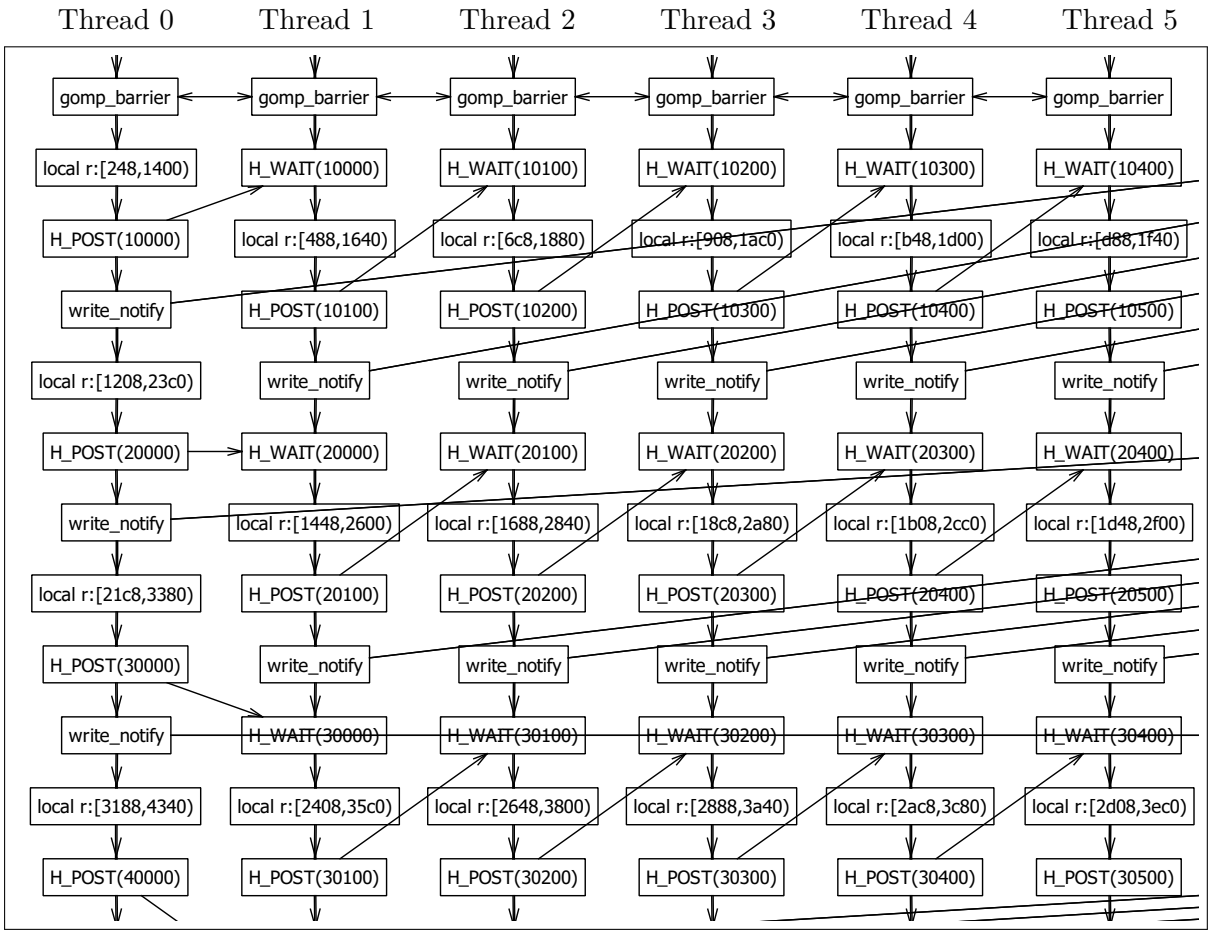


Abbildung 6.11: Task Graph des *async3d*-Codes (Ausschnitt von sechs Threads eines Prozesses)

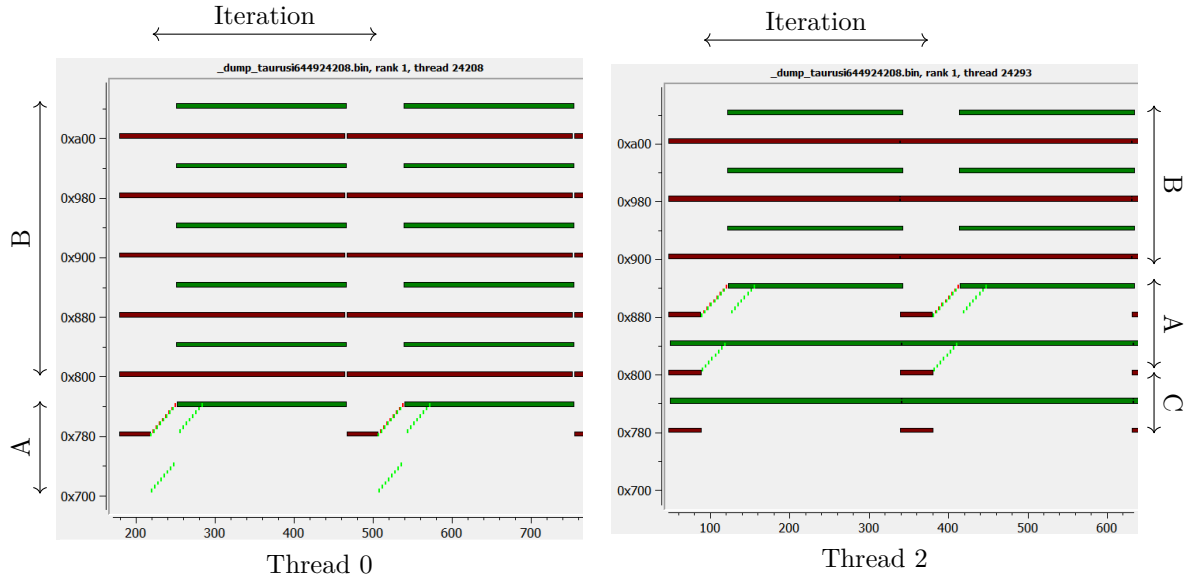


Abbildung 6.12: Gegenüberstellung der Speicherzugriffsdiagramme zweier Threads eines Prozesses von *async3d*

(B) Speicherbereich, in dem nachfolgende Threads ihre Berechnungen durchführen. Die Zugriffe in diesen Bereich sind für den dargestellten Thread nicht von Belang, asynchrone Schreibzugriffe werden deswegen nicht synchronisiert. Die asynchronen Lesezugriffe werden jedoch implizit synchronisiert, da das Versenden von Daten durch nachfolgende Threads von den eigenen Berechnungen abhängig ist.

(C) Speicherbereich, in dem vorhergehende Threads des eigenen Prozesses ihre Berechnungen durchführen. Die Ergebnisse dieser Berechnungen sind Voraussetzung für den dargestellten Thread, deswegen erfolgt eine implizite Synchronisation mit den Schreibzugriffen des vorherigen Prozesses. Die asynchronen Lesezugriffe dieser Threads müssen jedoch nicht synchronisiert werden.

Thread 0 bearbeitet Speicherbereich A, sobald die Daten vom vorhergehenden Prozess geschrieben wurden. Erkennbar ist dieses an den zwei dunkelroten Rechtecken in diesem Bereich, die mit Beginn der Iterationen enden. Die berechneten Daten am oberen Ende von Speicherbereich A sind für den nachfolgenden Prozess relevant und werden ab Zeitpunkt 240 und 530 an diesen gesendet. Entsprechende asynchrone Lesezugriffe sind im Diagramm dargestellt. Die anderen Threads benötigen ebenfalls Daten vom vorhergehenden Prozess und senden Ergebnisse an den nachfolgenden Prozess. Die dadurch von diesen Threads ausgelösten asynchronen Speicherzugriffe geschehen ebenfalls im Adressraum von Thread 0 (Speicherbereich B). Da diese jedoch nicht mit dem Adressbereich kollidieren, in dem Thread 0 lokal Daten verarbeitet, findet auch keine explizite Synchronisation mit diesen Zugriffen statt. Die Dauer dieser Zugriffe überspannt deswegen auch eine komplette Iteration. Am Ende einer Iteration werden diese Zugriffe nur indirekt durch OpenMP-Barrieren synchronisiert.
Speicherbereich A von Thread 2 überspannt einen anderen Adressraum. Da die Datenabhängigkeiten mehrdimensional sind, muss dieser Thread nicht nur auf den entsprechenden Thread des vorhergehenden Prozesses, sondern auch auf den vorhergehenden Thread des eigenen Prozesses warten. Das impliziert, dass der Thread auch auf alle asynchronen Schreibzugriffe warten muss, die in der aktuellen Iteration auf Adressbereiche vorhergehender Threads stattgefunden haben (Speicherbereich C). Im Speicherzugriffsdiagramm ist gut zu erkennen, dass alle asynchronen Schreibzugriffe auf die unteren Adressbereiche beendet sind, bevor die lokale Berechnung beginnt. Alle Speicherzugriffe, die oberhalb des Adressbereichs liegen, in dem Thread 2 operiert, werden nicht explizit synchronisiert und überspannen so jeweils eine gesamte Iteration. Die Analyse der Synchronisationsbeziehungen und Speicherzugriffe der *async3d*-Anwendung konnte keine Races feststellen. Das deckt sich mit dem bisher beobachteten Programmverhalten, bei dem die Validierung der Ergebnisse zuverlässig erfolgreich ist.

6.2.4 CFD-Berechnung über unstrukturierte Gitter

Strömungssimulationen werden in der Luftfahrt häufig über unstrukturierte Gitter vorgenommen [BKS+11]. Abbildung 6.13 zeigt das Gitter DLR-F6 [BE06], welches einen Flugzeugrumpf modelliert. Bereiche in der Nähe des Flugzeugrumpfes werden detailliert berechnet und sind daher sehr fein strukturiert. Für weiter entfernte Bereiche ist dagegen eine Berechnung auf einem groben Gitter ausreichend, wodurch Rechenzeit eingespart werden kann.

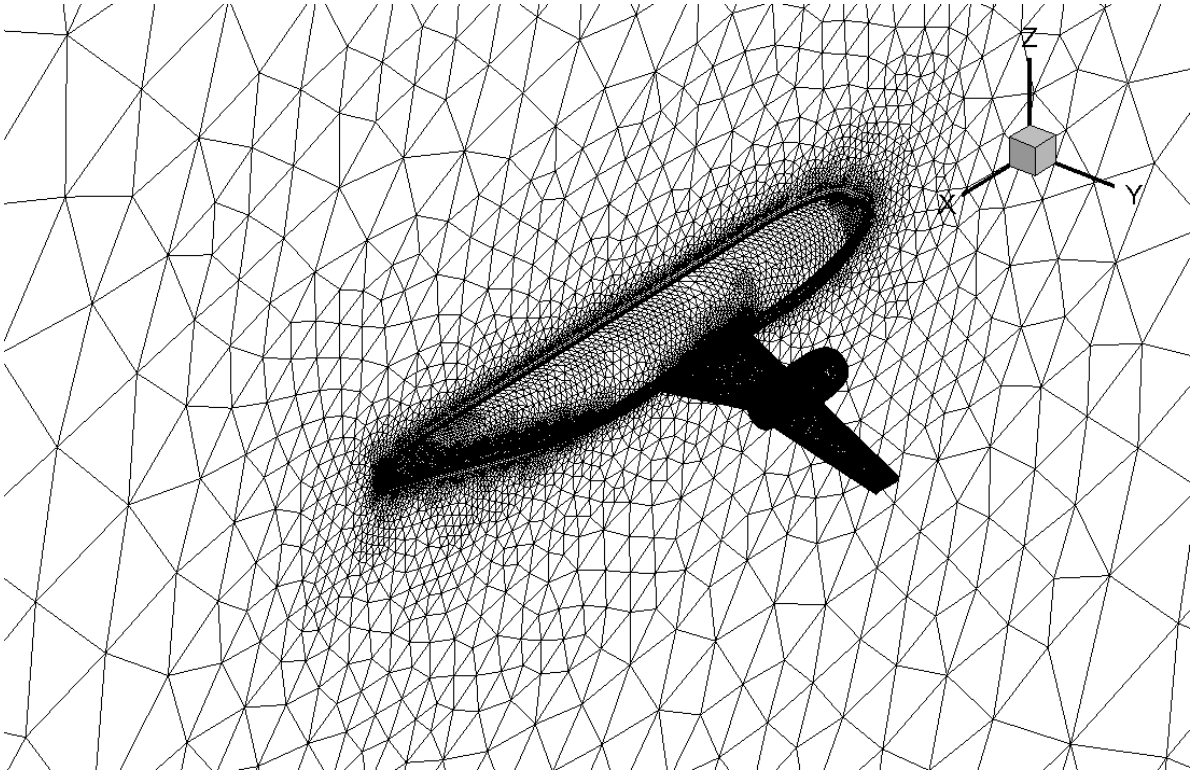


Abbildung 6.13: Ein für CFD-Berechnungen verwendetes unstrukturiertes Gitter

Der auf [Sim14a] bereitgestellte *cfdproxy* implementiert und untersucht die Effizienz verschiedener Parallelisierungs-APIs und -Strategien zum Austausch von Halos in unstrukturierten Gittern. Der Berechnungskern orientiert sich an dem vom DLR entwickelten Strömungslöser TAU [SGH06] zur Berechnung von Außenströmungen an Flugzeugen. Cfdproxy führt repräsentative strömungsmechanische Berechnungen durch und wird unter anderem im DLR als Teil der Evaluierungssuite für die Anschaffung von Hochleistungsrechnern eingesetzt. In dieser Arbeit wird die hybrid parallelisierte GASPI/OpenMP-Variante des Programms untersucht. Als Testdatensatz diente das DLR-F6-Gitter.

Auf Prozessebene wurden die üblichen GASPI-Synchronisationen implementiert. Auf Threadebene wurden Barrieren und die bereits im vorherigen Abschnitt erläuterten Fortschrittszähler verwendet. Darüber hinaus wurde auf dieser Ebene zur Synchronisation eine weitere Funktion `last_thread` verwendet. Diese Funktion wird an bestimmten Punkten in der Ausführung von jedem Thread eines Prozesses aufgerufen. Sie gibt `true` zurück, wenn es sich um den letzten Thread des Prozesses handelt, der diesen Ausführungspunkt erreicht. Die Erfassung dieser Funktion erfolgt durch Aufzeichnung des Aufrufs und des Rückgabewertes. Die Abbildung im Task-Graph-Modell erfolgt durch eine Überführung der Aufrufe in *POST* und *WAITCLEAR*-Operationen.

Ein Aufruf der `last_thread`-Funktion kann nur `true` liefern, nachdem alle anderen Threads diese Funktion am selben Ausführungspunkt betreten haben. Es bildet sich also eine happened-before-Relation von den Aufrufen, die `false` zurückgeben, zu dem Aufruf, der `true` zurückgibt. Deswegen wird jeder Aufruf, der `false` zurückgibt, als *POST*-Operation modelliert. Gibt ein Aufruf `true` zurück, so wird dieser als eine kollektive *WAITCLEAR*-Operation modelliert, die auf

alle *POST*-Operationen am Ausführungspunkt warten. Das erweiterte Replay-Verfahren beachtet dabei die entstehenden happened-before-Relationen und führt eine *WAITCLEAR*-Operation entsprechned erst aus, nachdem alle zugehörigen *POST*-Operationen ausgeführt wurden.
Die konkrete Implementierung der `last_thread`-Funktion in cfdproxy benutzt zur Identifizierung des Ausführungspunktes einen globalen Zähler. Dadurch wartet ein Thread in der `last_thread`-Funktion solange, bis alle anderen Threads den vorhergehenden Ausführungspunkt erreicht haben. Auch diese happened-before-Relationen wurden in das Replay-Verfahren integriert. Der Task Graph in Abbildung 6.14 zeigt die entstehenden Synchronisationsbeziehungen zwischen vier Threads über zwei Ausführungspunkte hinweg. Diese Beziehungen sind statisch nichtdeterministisch, denn an jedem Ausführungspunkt kann ein anderer Thread der *letzte* Thread sein und `true` zurückgeben.
Das Speicherzugriffsdiagramm in Abbildung 6.15 zeigt einen Thread von cfdproxy über mehrere Iterationen der Strömungsberechnung. Im oberen Teil des Diagramms wechseln sich entfernte Schreibzugriffe und direkte Lesezugriffe ab. Auffällig ist der untere Teil des Diagramms, in dem offensichtlich in jeder Iteration zwar PGAS-Lesezugriffe, nie jedoch Schreibzugriffe stattfinden. Ein solches Zugriffsmuster ist ein Hinweis auf einen Programmfehler, da ein wiederholtes Übertragen der Daten ohne zwischenzeitliche Schreibzugriffe redundant ist. Ein weiterer Hinweis ergibt sich aus dem Adressbereich der PGAS-Lesezugriffe, die laut Diagramm bei `0x0` beginnen. Die momentane Implementierung des Werkzeugs ermittelt die Adresse eines Segments implizit durch Auswertung eines `gaspi_segment_ptr`-Aufrufs. Einen solchen Aufruf muss jeder GASPI-Prozess durchführen, wenn er direkt auf PGAS-Segmente zugreifen will. Offenbar erfolgte für den Bereich, in dem die PGAS-Lesezugriffe stattfinden, kein `gaspi_segment_ptr`-Aufruf.
Bei der Untersuchung des Quellcodes konnte anhand der Hinweise aus dem Speicherzugriffsdiagramm schnell ein Programmierfehler lokalisiert werden. Ein Zeiger, der eigentlich auf ein GASPI-Segment zeigen sollte, verwies stattdessen auf einen nur lokal zugreifbaren Speicherbereich. Dadurch wurden die lokal berechneten Daten nicht an die für den Datenaustausch vorgesehene Stelle geschrieben, so dass auf diesen Bereich nur lesend, jedoch nie schreibend zugegriffen wurde.
Abbildung 6.16 zeigt das Speicherzugriffsdiagramm nach Korrektur dieses Programmfehlers. Aus diesem Diagramm geht hervor, dass die PGAS-Lesezugriffe genauso wie die PGAS-Schreibzu-

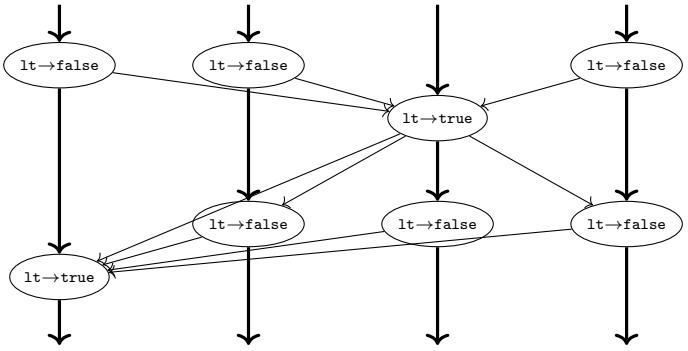


Abbildung 6.14: Synchronisationsbeziehungen von `last_thread`-Aufrufen (`lt`) aufgrund ihrer Rückgabewerte

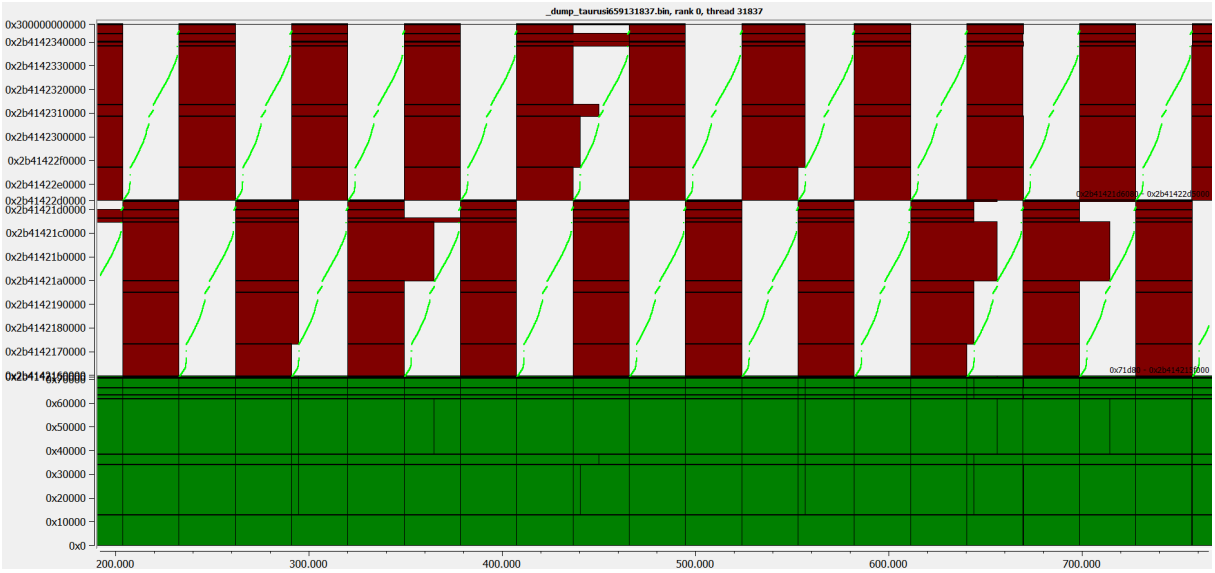


Abbildung 6.15: Speicherzugriffsdiagramm eines Threads des fehlerhaften cfdproxy

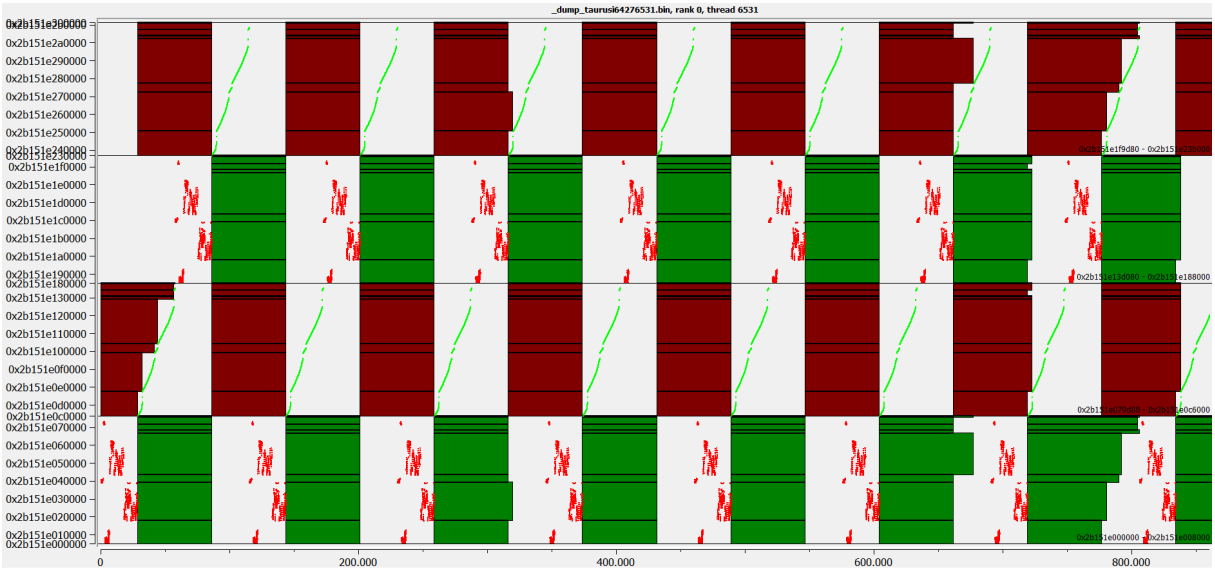


Abbildung 6.16: Speicherzugriffsdiagramm eines Threads des korrigierten cfdproxy

griffe abwechselnd auf zwei Segmente erfolgen. Da ein `gaspi_segment_ptr`-Aufruf für die Zeigerinitialisierung notwendig ist, konnten nun auch die Adressen der PGAS-Lesezugriffe aufgelöst werden. Es ergibt sich ein für Strömungssimulationen über unstrukturierte Gitter typisches Zugriffsmuster. Die direkten Speicherzugriffe sind abhängig von der lokalen Gitterstruktur. Die Indizes des Gitters sind durch einen Präprozessor-Schritt bereits sortiert, um die Speicherzugriffe möglichst lokal zu halten. Trotzdem sind die im Vergleich zu z. B. dem Stencil-Code irregulären Zugriffsmuster gut zu erkennen. Die PGAS-Speicherzugriffe richten sich nach den Halo-Bereichen der Gitterstruktur. Auch diese Bereiche sind nicht gleich verteilt. Jeder Prozess tauscht mit einer Reihe von Nachbarprozessen ein bestimmtes Datenvolumen aus, welches von dem jeweiligen Grenzbereich des Gitters abhängt. Die unterschiedlichen Größen der Datenvolumina sind anhand der gerahmt gezeichneten Blöcke für die PGAS-Speicherzugriffe erkennbar.

Dimension	≈Datenmenge (MB)	Dateigröße (MB)	original Zeit (s)	instrumentiert Zeit (s)	Verlangsamung instr./orig.
500	3.000	842	0,20	8,77	43,85
600	5.184	1.824	0,31	16,02	51,68
700	8.232	2.429	0,49	22,14	45,18
800	12.288	3.423	0,73	34,55	47,33
900	17.496	5.660	1,20	52,72	43,93
1000	24.000	8.001	2,72	86,40	31,76

Tabelle 6.2: Aufwand für die Aufzeichnung einer Matrixmultiplikation mit Memaccessrecord

6.3 Quantitative Evaluierung der prototypischen Realisierung

Das Ziel der quantitativen Untersuchung ist die Feststellung der Skalierbarkeit der in dieser Arbeit eingeführten Verfahren. Der Begriff der Skalierbarkeit ist unterschiedlich definierbar [Hil90, DRW06]. In dieser Arbeit wird ein Algorithmus als skalierbar bezogen auf eine bestimmte Problemgröße bezeichnet, wenn seine Effizienz nicht wesentlich von dieser Größe abhängt. Ein solcher Algorithmus sollte also annähernd lineare Laufzeitkomplexität aufweisen. Als relevante Problemgrößen werden die Anzahl der analysierten Threads ($|P|$) und die Anzahl der aufgezeichneten Programmereignisse bzw. Tasks ($|T|$) untersucht.
Zusätzlich zu den in Tabelle 6.1 genannten Anwendungen wird außerdem das Programm *heat* in die quantitative Untersuchung mit einbezogen. Dieses Programm implementiert einen zweidimensionalen Stencil-Code. Es wurde in die Evaluierung aufgenommen, da es nach jeder Iteration eine kollektive Reduktion durchführt. Wie bereits in [KMPN16] festgestellt wurde, beeinflussen kollektive Operationen die Komplexität des Replay-Verfahrens.
Alle Trendlinien-Formeln wurden mit Microsoft® Excel® 2013 ermittelt. Der Wert R^2 in diesen Tabellen gibt das Bestimmtheitsmaß an.

6.3.1 Zeit- und Speicherbedarf für die Aufzeichnung von direkten Speicherzugriffen

Tabelle 6.2 zeigt Zeiten und Dateigrößen für die Aufzeichnung einer typischen Matrixmultiplikation mit Memaccessrecord. Die innere Schleife einer solchen Multiplikation produziert sehr schnell sehr viele Speicherzugriffe. Dadurch ist diese Anwendung ein Grenzfall für das Laufzeitverhalten von Memaccessrecord, da das Erfassen und Speichern der Zugriffsadressen die Ausführung des eigentlichen Programms dominiert. Die geschätzte anfallende theoretische Datenmenge ist $D = n^3 * \texttt{sizeof(ADDR)} * 3$ mit n als Matrixdimension und `sizeof(ADDR)` als Größe eines `TRACE ID`- bzw. `ADDR`-Eintrages (auf Taurus 8 Bytes). Die innere Schleife produziert in jeder Iteration 3 Einträge (eine `TRACE ID` und 2 Speicherzugriffe). Die angegebenen Zeiten und Dateigrößen sind jeweils Minimalwerte aus mehreren Testläufen. Die Zeiten sind der Wert *real* des Befehls `time`. Diese *wall clock times* beinhalten also auch alle Kosten, die durch die Instrumentierung am Beginn eines Programmlaufs anfallen. Die Dateigrößen spiegeln im Verhältnis zur Datenmenge die Effizienz der Komprimierung wider, die unabhängig von der Matrixgröße eine Reduktion auf ca. 30% erreicht. Die bis zu 50-fache Verlangsamung sinkt dagegen mit steigender Problemgröße, da der Einfluss der Start-Instrumentierung auf die Laufzeit abnimmt. Im Gegensatz zu diesem Extremfall sind die Kosten für die Aufzeichnung von GASPI-Anwen-

Testfall	Prozesse	Dateigrößen Summe (MB)	original Zeit (s)	instrumentiert Zeit (s)	Verlangsamung instr./orig.
stencil (40.000)	8	22.891	9,87	59,66	6,04
	16	23.159	6,92	40,68	5,88
	32	25.417	10,92	34,90	3,19
stencil (80.000)	8	45.732	10,54	121,6	11,5
	16	46.264	8,43	76,17	9,03
	32	50.829	12,46	50,62	4,06
cfdproxy	12	6.438	10,47	52,9	5,05
	24	10.047	10,75	49,02	4,56
	192	37.179	30,76	61,84	2,01

Tabelle 6.3: Aufwand für die Aufzeichnung von GASPI-Anwendungen mit Memaccessrecord

dungen wesentlich niedriger. Tabelle 6.3 zeigt Zeiten und Dateigrößen für Testläufe mit *stencil* und *cfdproxy*. Für den Test von *cfdproxy* wurde das F6-Gitter als Problem auf verschiedenen Prozesszahlen mit entsprechend angepasster Partitionierung des Gitters gerechnet. Mit *stencil* wurden 40.000 und 80.000 Iterationen auf einem 1024x16 Zellen großem Feld berechnet, das über die Prozesse verteilt wurde.

Die angegebenen Zeiten sind jeweils die *wall clock time* des am längsten laufenden Prozesses. Die Unterschiede zu den Laufzeiten der anderen Prozesse sind sehr gering und betragen in allen Fällen weniger als eine Sekunde, da am Ende der Prozesse jeweils kollektive Reduktionen zur Berechnung des Gesamtergebnisses ausgeführt werden. Für alle Testfälle sinkt die Verlangsamung bei steigenden Prozesszahlen, da das Verhältnis von Rechenlast und damit der Anzahl direkter Speicherzugriffe zur Kommunikationslast für einen einzelnen Prozess bei gleichbleibender Problemgröße sinkt. Die Verlangsamung von *stencil* ist trotzdem signifikant. Das liegt zum einen an der effizienten Kommunikationsstrategie, zum anderen auch an der sehr einfachen Berechnungsvorschrift für eine Zelle. Dadurch kommt diese Anwendung dem Verhalten der Matrixmultiplikation sehr nahe und kann als Grenzfall für GASPI-Anwendungen betrachtet werden. Für *cfdproxy* erhöht sich die Laufzeit durch die Aufzeichnung nur um das maximal Fünffache.

Da jeder Prozess eine eigene Aufzeichnungsdatei generiert, ist in Tabelle 6.3 die Gesamtmenge der produzierten Daten als Summe der Größen der Einzeldateien angegeben. In jeder Datei werden zusätzlich zu den aufgezeichneten Programmereignissen auch die statischen Adress- und Funktions-Informationen gespeichert. Außerdem finden mit zunehmender Prozessanzahl insgesamt mehr Kommunikationsereignisse statt. Aus diesen Gründen steigt die Datenmenge mit zunehmender Prozessanzahl. Die Anzahl der direkten Speicherzugriffe ist dagegen vor allem von der Problemgröße abhängig. Die Datenmenge steigt weit weniger stark an als die Anzahl der Prozesse. Daraus kann gefolgert werden, dass der Großteil der Datenmenge aus aufgezeichneten direkten Speicherzugriffen besteht. Die Aufzeichnung wird also weder bei längeren Laufzeiten noch bei steigenden Prozesszahlen ineffizienter.

Durch die signifikante Verlangsamung während der Aufzeichnung kann die Erfassung von direkten Speicherzugriffen mit Memaccessrecord für realitätsnahe Echtzeitmessungen nicht genutzt werden. Aus diesem Grund ist es wichtig, dass für die in dieser Arbeit entwickelten Auswertungsmethoden die tatsächliche Reihenfolge der aufgezeichneten Programmereignisse keine Rolle

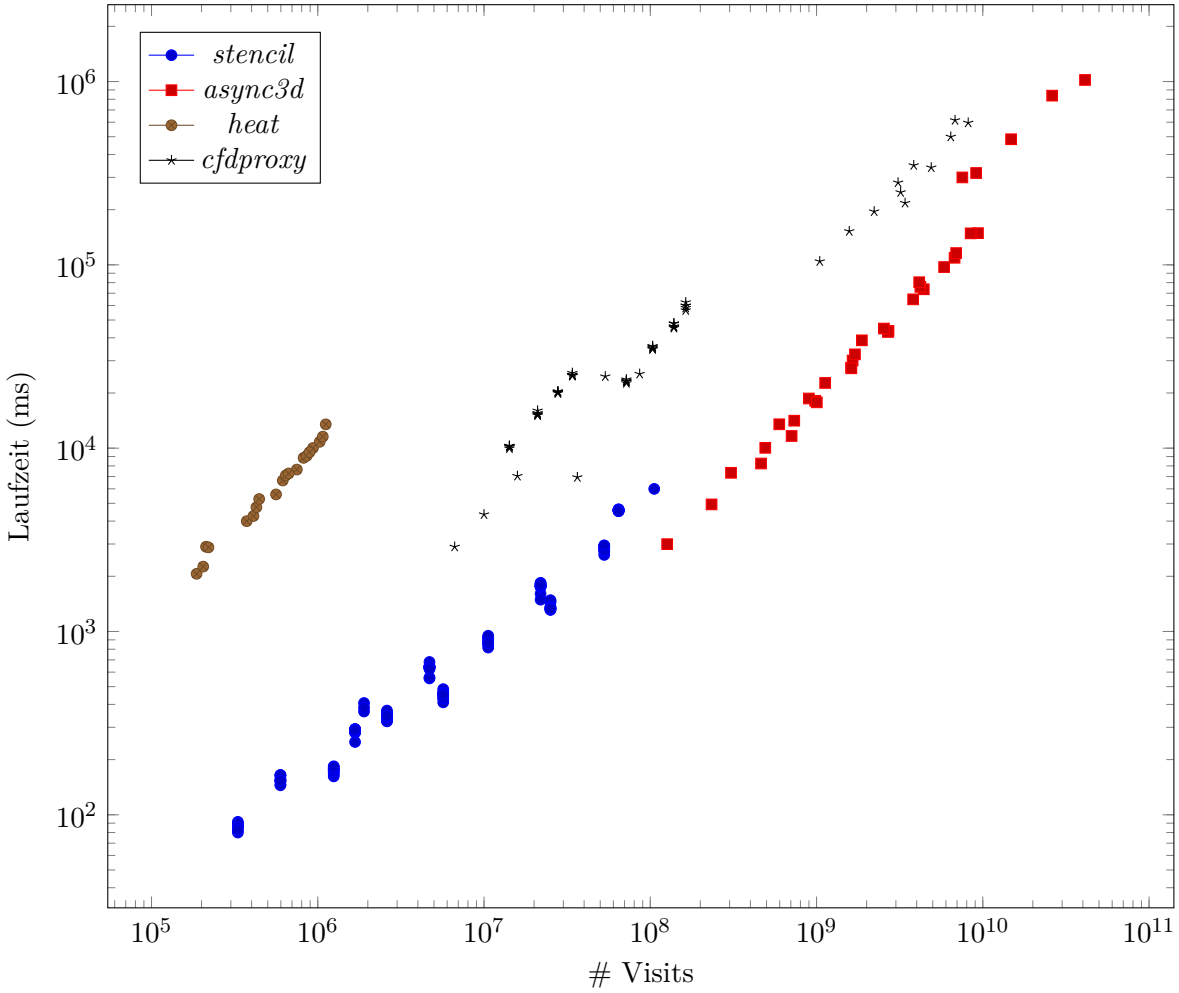


Abbildung 6.17: Verhältnis von Visits und Laufzeit des Replay-Verfahrens

spielt, so dass nur die logischen Beziehungen zwischen diesen Ereignissen untersucht werden. Bisherige Arbeiten auf diesem Gebiet haben dieses Problem trotz ähnlicher Verlangsamungswerte z. B. bei der Berechnung von Synchronisationsbeziehungen außer Acht gelassen.
Andere Aufzeichnungswerkzeuge für direkte Speicherzugriffe geben Verlangsamungen von 2,5 (MC-Checker, Abschnitt 3.2.2) bis zu 100 (Cachegrind, Abschnitt 3.2.3) an. Die durch Memaccessrecord verursachte Verlangsamung von 2 bis 11 für praktische Anwendungsfälle befindet sich am unteren Ende dieser Skala und ist für Messungen in der Praxis gut vertretbar.

6.3.2 Untersuchung der Laufzeit des Replay-Verfahrens

In Abschnitt 4.1.4 (Seite 42) wurde die Komplexität des Replay-Verfahrens bereits mit kleiner als $\mathcal{O}(|T|^2)$ bezogen auf die Anzahl $|T|$ der untersuchten Tasks angegeben. Die folgende Untersuchung der tatsächlichen Komplexität des Verfahrens misst die Laufzeit der Funktion `replay_tasks` (Listing 4 (Seite 41) über alle Threads und die Gesamtanzahl der während der Erreichbarkeitstests durchlaufenen Tasks (*Visits*). Diese Werte werden in Relation zur Gesamtanzahl der Tasks und zur Anzahl der Threads gesetzt. Für homogen parallelisierte Programme sind die Anzahl der Threads und die Anzahl der Prozesse gleich.

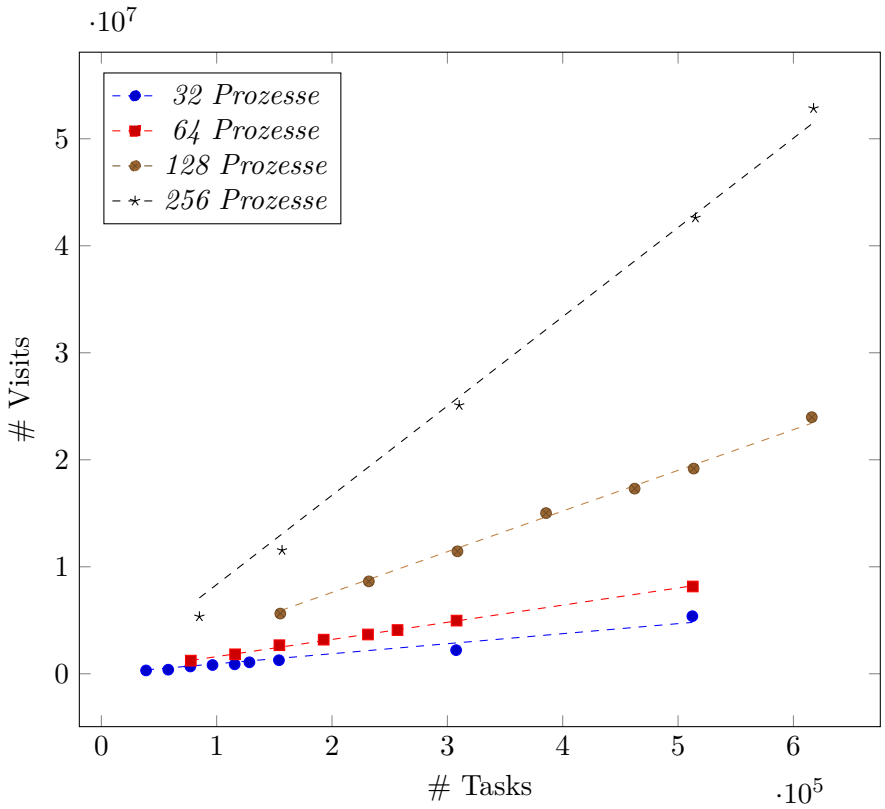


Abbildung 6.18: Lineare Komplexität des Replay-Verfahrens für die Anwendung *stencil*

$\|P\|$	Formel	R^2
32	$y = 9,3744x$	0,9542
64	$y = 16,042x$	0,9983
128	$y = 38,023x$	0,9966
256	$y = 83,423x$	0,9951

Tabelle 6.4: Lineare Trendlinien-Formeln für Abbildung 6.18 (x=# Tasks, y=# Visits)

Abbildung 6.17 zeigt für die in dieser Arbeit analysierten Anwendungen das Verhältnis der Laufzeit zur Anzahl der Visits (#Visits). Für jede Anwendung wurden verschiedene Problemgrößen und Prozesszahlen erfasst, so dass sich für das Replay-Verfahren mehrere Visit-Zahlen pro Anwendung ergeben. Für jeden erfassten Programmlauf wurde die Laufzeit des Replay-Verfahrens dann mehrmals gemessen. Aus der Messung geht hervor, dass die Anzahl der Visits tatsächlich das für die Laufzeit des Replay-Verfahrens entscheidende Kriterium darstellt. Das Verhältnis von Laufzeit zur Anzahl der Visits variiert je nach Anwendungsklasse und den damit behandelten Synchronisationsoperationen, ist jedoch innerhalb einer Klasse konstant. Aus diesem Grund wird im Folgenden die Anzahl der Visits zur weiteren Evaluierung des Replay-Verfahrens betrachtet, da dieser Wert im Gegensatz zur tatsächlichen Laufzeit von äußeren Faktoren unabhängig ist.
Für die homogen parallelisierte *stencil*-Anwendung zeigt Abbildung 6.18 die Anzahl der Visits im Verhältnis zur Anzahl der Tasks $|T|$ für verschiedene Threadzahlen $|P|$. Für jede Threadzahl besteht ein linearer Zusammenhang zwischen $|T|$ und der Anzahl der Visits. Der Gradient der linearen Trendlinien hängt ebenfalls linear von $|P|$ ab (Tabelle 6.4). Damit ergibt sich die Komplexität des Replay-Verfahrens für die *stencil*-Anwendung zu $\mathcal{O}(|T| * |P|)$.

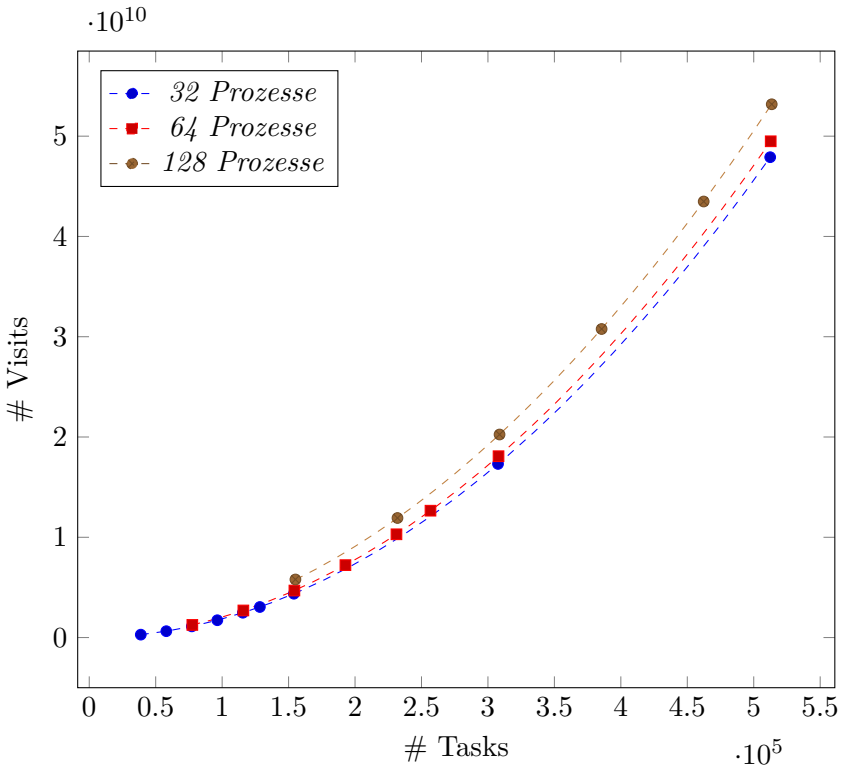


Abbildung 6.19: Quadratische Komplexität des Replay-Verfahrens für die *stencil*-Anwendung bei abgeschalteter topologischer Sortierung

Diese Komplexität ist typisch für Programme, die Halodaten mit Nachbar-Ranks austauschen. Während der Pfadsuche traversiert der Erreichbarkeitstest aufgrund der topologischen Sortierung nur wenig in die Tiefe des Graphen. Allerdings sind an jedem Synchronisationspunkt mit einem Nachbar-Rank zwei alternative Suchoptionen vorhanden: entweder weiter auf dem aktuellen Rank oder Abzweigen zum Nachbar-Rank. In der *stencil*-Anwendung hat jeder Rank zwei Nachbar-Ranks. Da die Pfadsuche den Zielrank nicht beachtet, wird sie in ca. 50% der Fälle auf den falschen Nachbar-Rank abbiegen und von dort aus dann die gesamte Breite des Graphen traversieren. Aus diesem Grund ergibt sich die Abhängigkeit der Komplexität von $|P|$.

Dass die lineare Komplexität bezogen auf $|T|$ durch die Optimierung mittels topologischer Sortierung erreicht wurde, verdeutlicht Abbildung 6.19. Dieses Diagramm zeigt die Anzahl der Visits

$\|P\|$	Formeltyp	Formel	R^2
32	Linear	$y = 22414x$	$0,8225$
	Polynomial	$y = 0,1816x^2 + 419,96x$	1
	Potenz	$y = 0,2653x^{1,9693}$	1
64	Linear	$y = 46580x$	$0,8275$
	Polynomial	$y = 0,1845x^2 + 1909,9x$	1
	Potenz	$y = 0,4394x^{1,9333}$	1
128	Linear	$y = 98156x$	$0,8374$
	Polynomial	$y = 0,1854x^2 + 8377,7x$	1
	Potenz	$y = 1,2013x^{1,864}$	$0,9998$

Tabelle 6.5: Trendlinien-Formeln für Abbildung 6.19 (x=# Tasks, y=# Visits)

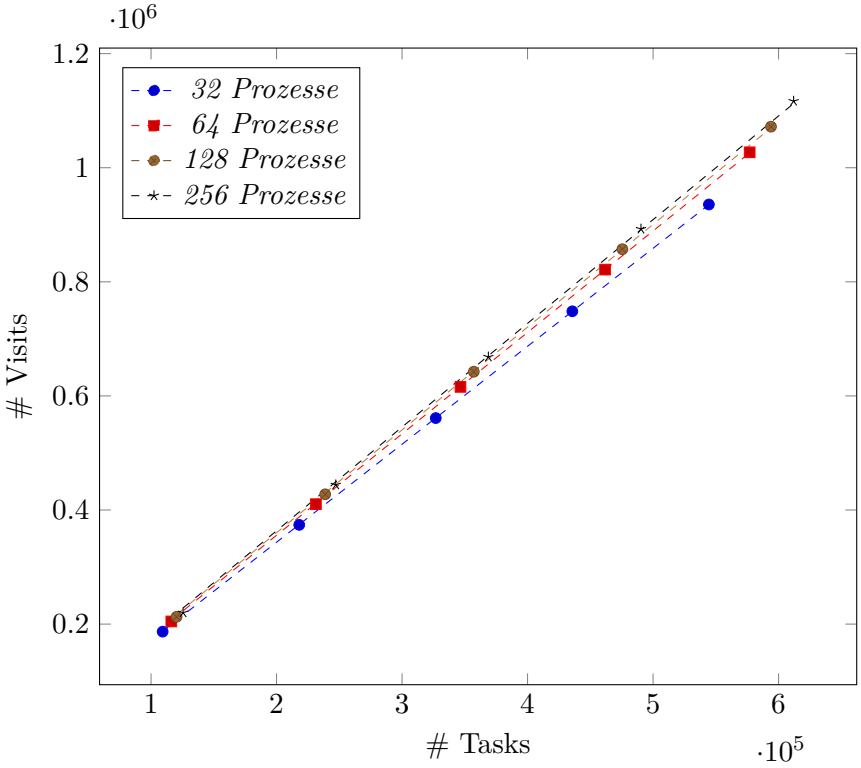


Abbildung 6.20: Weitgehende Unabhängigkeit der Komplexität des Replay-Verfahrens in Bezug auf die Anzahl der Prozesse für die *heat*-Anwendung

$\|P\|$	Formel	R^2
32	$y = 1,7175x$	1
64	$y = 1,7784x$	1
128	$y = 1.8014x$	0,9999
256	$y = 1,818x$	0,9998

Tabelle 6.6: Lineare Trendlinien-Formeln für Abbildung 6.20 (x=# Tasks, y=# Visits)

für einige Fälle aus Abbildung 6.18, wobei der Erreichbarkeitstest ohne Beachtung der topologischen Sortierung durchgeführt wurde. Die Untersuchung bestätigt die Eingangsbetrachtungen zur Komplexität des Verfahrens in Abschnitt 4.1.4 (Seite 42). Das Abschalten der Optimierung führt zu einer quadratischen Laufzeitkomplexität $\mathcal{O}(|T|^2)$ (Tabelle 6.5). Daraus resultiert in den betrachteten Fällen für $5 * 10^5$ Tasks eine um mindestens ca. 2500-fache höhere Visit-Anzahl und Ausführungsdauer.

Für Programme mit vielen kollektiven Operationen wie z. B. *heat* wurde in [KMPN16] eine Komplexität von $\mathcal{O}(|T| * |P|^2)$ ermittelt. Inzwischen wurde die Behandlung von kollektiven Synchronisationsoperationen im Erreichbarkeitstest dahingehend optimiert, dass bei der Tiefensuche Pfade bevorzugt werden, die direkt zum Zielprozess bzw. -thread führen. Mit dieser Heuristik konnte die Komplexität des Verfahrens für diese Anwendungsklasse verringert werden. Abbildung 6.20 zeigt die Anzahl der Visits für *heat*. Die Gradienten der Trendlinien sind fast unabhängig von $|P|$ (Tabelle 6.6). Allerdings wird in der momentanen Implementierung der passende Zielprozess binär gesucht. Diese Suche verändert nicht die Anzahl der Visits, benötigt aber zusätzliche Zeit. Daraus ergibt sich eine erhöhte Laufzeit in Bezug auf die Anzahl der Visits.

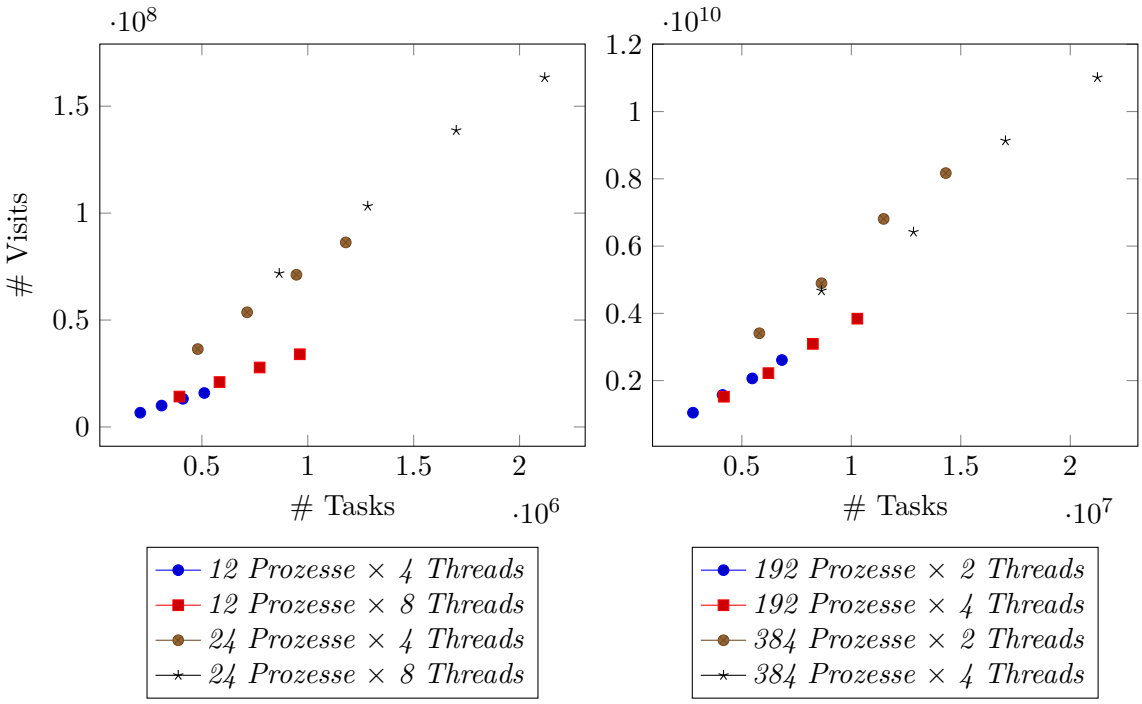


Abbildung 6.21: Komplexität des Replay-Verfahrens für die hybrid parallele *cfdproxy*-Anwendung

$\|P\|$	Formel	R^2
12 Prozesse × 4 Threads	$y = 31,643x$	0,9966
12 Prozesse × 8 Threads	$y = 35,703x$	0,999
24 Prozesse × 4 Threads	$y = 74,311x$	0,9976
24 Prozesse × 8 Threads	$y = 79,439x$	0,9901
192 Prozesse × 2 Threads	$y = 380,3x$	0,9996
192 Prozesse × 4 Threads	$y = 370,78x$	0,9968
384 Prozesse × 2 Threads	$y = 577,98x$	0,9959
384 Prozesse × 4 Threads	$y = 522,17x$	0,9929

Tabelle 6.7: Lineare Trendlinien-Formeln für Abbildung 6.21 (x=# Tasks, y=# Visits)

Das wird in Abbildung 6.17 anhand des nach oben verschobenen Korridors für die Messpunkte von *heat* deutlich.

Auch für hybrid parallelisierte Anwendungen zeigt das Replay-Verfahren lineare Laufzeitkpomplexität $\mathcal{O}(|T|)$. Abbildung 6.21 zeigt das Verhalten von *cfdproxy* für verschiedene Kombinationen von Thread- und Prozessanzahl. Auf Threadebene finden vor allem kollektive Synchronisationen statt. Deswegen ist die Laufzeit ähnlich wie bei *heat* von der Threadanzahl pro Prozess fast unabhängig. Die Synchronisationsstrategie auf Prozessebene entspricht eher einer Stencil-Berechnung, was wiederum wie für *stencil* in einer linearen Abhängigkeit von der Prozessanzahl resultiert (Tabelle 6.7).

Zusammenfassend wurde die in Abschnitt 4.1.4 (Seite 42) postulierte quasi-lineare Laufzeit bezogen auf $|T|$ durch die Evaluierung bestätigt. Für alle hier untersuchten Anwendungen ergaben sich sogar lineare Laufzeiten. Außerdem konnte die Evaluierung zeigen, dass die topologische Sortierung ein entscheidender Faktor zum Erreichen linearer Laufzeit ist. Das Replay-Verfahren

skaliert auch mit der Anzahl der Threads und Prozesse. Je nach verwendetem Synchronisationsverfahren ist die Laufzeit linear bezogen auf $|P|$ oder sogar fast unabhängig von dieser Größe.

6.3.3 Untersuchung der Laufzeit der Data-Race-Analyse

Das in Abschnitt 5.2.2 (Seite 72) skizzierte Verfahren zum Finden von data races besteht aus zwei Schritten. Im ersten Schritt werden den PGAS-Speicherzugriffen Zeitstempel entsprechend den Formeln 5.1 und 5.2 (Seite 70) zugewiesen. Das Ergebnis dieser *Zeitstempelung* wird auch zur Darstellung der PGAS-Speicherzugriffe im Speicherzugriffsdiagramm benötigt. Im zweiten Schritt, dem *Adressvergleich*, werden die Adressintervalle der sich zeitlich überlagernden direkten Speicherzugriffe und der PGAS-Speicherzugriffe verglichen. Beide Schritte werden im Folgenden separat betrachtet.
Der Algorithmus zur Zeitstempelung ist in Listing 15 schematisch für einen Thread dargestellt. Für jeden Task eines Threads wird dessen Zeitstempel zweimal durch den gesamten Task Graphen propagiert. Im ersten Durchlauf wird der Task Graph vom Task `e` aus vorwärts traversiert und die Endzeiten aller erreichten PGAS-Speicherzugriffe werden auf den Zeitstempel von `e` gesetzt. Die Traversierung wird jeweils an bereits früher erreichten Tasks abgebrochen. Im zweiten Durchlauf wird der Task Graph rückwärts traversiert und die Startzeiten aller erreichten PGAS-Speicherzugriffe werden auf den Zeitstempel von `e` gesetzt. Auch hierbei werden nur nicht bereits behandelte Tasks traversiert. Am Ende dieser zwei Durchläufe sind die Start- und Endzeiten der PGAS-Speicherzugriffe auf Zeitstempel entsprechend den Formeln 5.1 und 5.2 bezüglich der Tasks eines Threads `t` gesetzt. Dieses Ergebnis wird zur Anzeige des Threads in einem Speicherzugriffsdiagramm genutzt oder zum zweiten Schritt der Data-Race-Analyse, dem Adressvergleich, weitergereicht.
In der Funktion `timestamping` wird jeder Task des Task Graphen genau zweimal durchlaufen (in jeder Schleife einmal), da die Traversierung bei bereits durchlaufenen Tasks abbricht. Das gesamte Verfahren durchläuft damit für jeden Thread den gesamten Task Graphen, es hat also eine Komplexität von $\mathcal{O}(|P| * |T|)$. Im Gegensatz zum Replay-Verfahren ist diese Komplexität unabhängig von den im analysierten Programm verwendeten Synchronisationsverfahren, da keine Heuristiken eingesetzt werden.

```
1  function timestamping (Thread t)
2  {
3    for_all (e : tasks of t)
4      propagate timestamp(e) to all PGAS accesses
5      reaching e as end time
6
7    reverse_for_all (e : events of t)
8      propagate timestamp(e) to all reachable PGAS accesses
9      from e as start time
10 }
```

Listing 15: Algorithmus zur Zeitstempelung

Der Algorithmus zum Adressvergleich vergleicht für jeden Thread die Adressen der PGAS-Zugriffe auf den Speicherbereich des zugehörigen Prozesses mit den direkten Speicherzugriffen

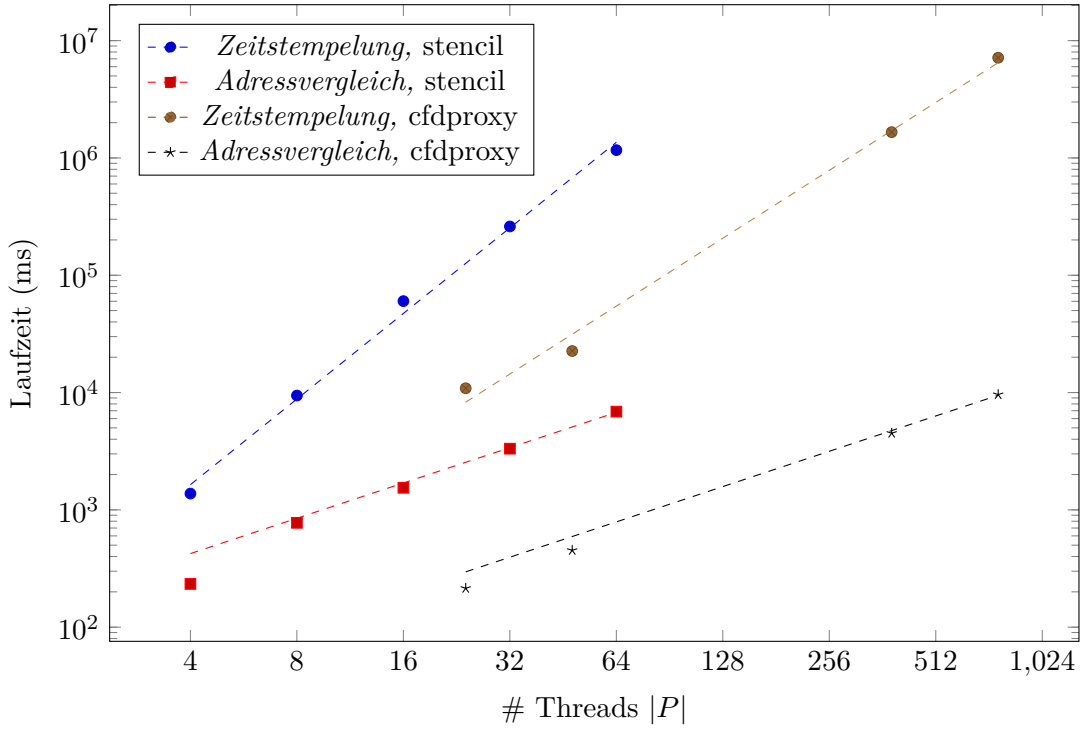


Abbildung 6.22: Komplexität der Schritte *Zeitstempelung* und *Adressvergleich* als Teile der Data-Race-Analyse

des Threads. Dabei werden für jeden PGAS-Zugriff nur die direkten Speicherzugriffe betrachtet, die innerhalb des durch die Zeitstempelung ermittelten Intervalls liegen. Da die Zeitstempel als Indizes auf die Tasks des Threads verwendet werden können, ist die Ermittlung der zu testenden Speicherzugriffe in konstanter Zeit möglich. Auch ist die Menge der jeweils zu testenden Speicherzugriffe klein in Bezug auf die Anzahl der Tasks $|T|$ und unabhängig von dieser Größe. Die Laufzeit des Adressvergleichs ist damit vor allem von der Anzahl der PGAS-Speicherzugriffe abhängig. Die Anzahl dieser Zugriffe ist für viele praktische Anwendungen linear von $|T|$ abhängig. Damit ergibt sich für den Adressvergleich eine Komplexität von $\mathcal{O}(|T|)$.

Die in Abbildung 6.22 gezeigten Zeitmessungen für die Anwendungen *stencil* und *cfdproxy* bestätigen die soeben ausgeführten Überlegungen zur Komplexität. Für beide Anwendungen wurde die jeweils gleiche Problemgröße für verschiedene Prozesszahlen gerechnet. Während der Data-Race-Analyse wurden Zeitstempelung und Adressvergleich separat gemessen.

Die Zeitstempelung zeigt für beide Anwendungen eine quadratische Laufzeit bezogen auf $|P|$ (Tabelle 6.8). Der Grund dafür ist der datenparallele Charakter beider Anwendungen. Für solche Anwendungen sinkt mit steigendem $|P|$ die Anzahl der direkten Speicherzugriffe pro Thread, da das lokal zu berechnende Problem kleiner wird. Allerdings sind die direkten Speicherzugriffe in den gemeinsamen Speicherbereich für diesen Anwendungstyp aufeinanderfolgend. Die damit entstehenden Blöcke von direkten Speicherzugriffen werden unabhängig von der Anzahl der Zugriffe zu einem Task zusammengefasst. Dadurch wird die Anzahl der Tasks pro Thread bei sonst gleicher Problemgröße fast unabhängig von $|P|$. Die Gesamtzahl $|T|$ der Tasks über alle Threads wird also linear abhängig von $|P|$ und es gilt $|T| = x * |P|$. Unter Anwendung dieser Gleichung kann die Komplexität der Zeitstempelung für datenparallele Anwendungen zu

$\|P\|$	Formeltyp	Formel	R^2
stencil (Zeitstempelung)	Linear	$y = 15392x$	$0,8599$
	Polynomial	$y = 308,37x^2 - 1540x$	$0,999$
	Potenz	$y = 57,065x^{2,424}$	$0,996$
stencil (Adressvergleich)	Linear	$y = 106,06x$	$0,9973$
	Polynomial	$y = 0,186x^2 + 95,865x$	$0,999$
	Potenz	$y = 54,068x^{1,1861}$	$0,9879$
cfdproxy (Zeitstempelung)	Linear	$y = 8279,5x$	$0,9094$
	Polynomial	$y = 12,926x^2 - 622,65x$	1
	Potenz	$y = 18,367x^{1,9234}$	$0,9936$
cfdproxy (Adressvergleich)	Linear	$y = 12,345x$	$0,9983$
	Polynomial	$y = 0,0022x^2 + 10,815x$	$0,999$
	Potenz	$y = 6,4722x^{1,0993}$	1

Tabelle 6.8: Trendlinien-Formeln für Abbildung 6.22 (x=#Prozesse, y=Laufzeit)

$\mathcal{O}(|P|^2)$ vereinfacht werden. Laut Tabelle 6.8 liegt die Laufzeit für *stencil* möglicherweise sogar oberhalb quadratischer Komplexität. Ein Grund dafür kann sein, dass die Anzahl der während einer Traversierung erreichten, jedoch bereits vorher besuchten Tasks nicht in die Komplexitätsbetrachtungen einbezogen wurden.

Die evaluierte Realisierung des Adressvergleichs liest die direkten Speicherzugriffe nur bei Bedarf aus der als Datei vorliegenden Aufzeichnung, da die Menge dieser Zugriffe potentiell sehr groß ist. Trotz der dadurch notwendigen Dateizugriffe ist die Laufzeit des Adressvergleichs gerade für größere Prozesszahlen wesentlich niedriger als die Laufzeit der Zeitstempelung. Denn wenn wie im vorherigen Absatz $|T| = x * |P|$ angenommen wird, dann ergibt sich für den Adressvergleich eine lineare Komplexität von $\mathcal{O}(|P|)$.

Aus dieser Evaluierung der Data-Race-Analyse muss geschlussfolgert werden, dass die Optimierung der Zeitstempelung eine wichtige weiterführende Aufgabe darstellt, um das Verfahren skalierbarer zu machen. Der im Prototyp verwendete Algorithmus ist zwar einfach zu realisieren, traversiert aber immer über alle Tasks des Task Graphen. Da aber nur die Zeitstempel der PGAS-Speicherzugriffe, die im PGAS-Speicher des betrachteten Threads liegen, von Belang sind, ist in der Funktion `timestamping` ein Iterieren über diese Zugriffe statt zweimal über alle Tasks eines Threads angebracht. In dem Fall muss ein Algorithmus gefunden werden, der für einen PGAS-Speicherzugriff die begrenzenden Tasks des betrachteten Threads findet. Hierbei kann die topologische Sortierung hilfreich sein. Unter Umständen ist auch die Durchführung einer reversen topologischen Sortierung, wie sie in [VCJZ14] beschrieben wird, notwendig.

6.4 Schlussfolgerungen

Die qualitative Evaluierung zeigt, dass die in dieser Arbeit entwickelten Analysemethoden auf vielfältige Weise zur Untersuchung von GASPI-Programmen geeignet sind. Speicherzugriffsdiagramme sind sehr zweckmäßig, um data races oder Optimierungspotential in PGAS-Programmen zu finden und zu verstehen. Darüber hinaus hat sich gezeigt, dass auch andere Programmfehler wie z. B. das Lesen uninitialisierter Variablen durch Speicherzugriffsdiagramme sichtbar werden.

Weiterhin wurde gezeigt, wie das zugrunde liegende Task-Graph-Modell um benutzerdefinierte Synchronisationsoperationen erweiterbar ist. Dadurch können viele hybrid parallele PGAS-Programme aus der Anwendungspraxis einer Analyse zugänglich gemacht werden. Für solche Programme war bisher kein entsprechendes Werkzeug verfügbar.
Die quantitative Evaluierung ergibt ein differenziertes Bild der prototypisch implementierten Verfahren. Die Aufzeichnung von direkten Speicherzugriffen führt zu einer signifikanten Laufzeiterhöhung, wodurch eventuell erfasste Echtzeitdaten einer starken Verfälschung unterworfen wären. Die in dieser Arbeit eingeführten Methoden analysieren einen Programmlauf jedoch auf der rein logischen Ebene, so dass die Echtzeitdaten nicht von Belang sind. Insgesamt bleibt die Verlangsamung von GASPI-Programmen in einem vertretbaren Rahmen und stellt keine Hürde für die Durchführung einer Analyse dar.
Die durch die Aufzeichnung produzierten Datenmengen erreichen trotz der Komprimierung schon für kurze Läufe von GASPI-Programmen mehrere Gigabyte. Aus diesem Grund ist diese Aufzeichnungsmethode vor allem für die Analyse kurzer Programmläufe geeignet. Für die Untersuchung lang laufender Prozesse ist eine on-the-fly-Implementierung der beschriebenen Methoden besser geeignet. Allerdings sind die Analysemethoden robust gegenüber der Anzahl der untersuchten Threads und der Länge des Programmlaufs, so dass kurze Testläufe über ein repräsentatives Problem meist ausreichend sind.
Das Replay-Verfahren wurde durch den Einsatz der topologischen Sortierung so optimiert, dass die Komplexität praktisch linear bezogen auf die Anzahl der untersuchten Tasks ist. Damit wird die praktische Einsatzfähigkeit des Verfahrens erreicht. Insbesondere konnte im Vergleich zu früheren Verfahren eine exponentielle Komplexität vermieden werden.
Der bisher implementierte Algorithmus zur Zeitstempelung für einen einzelnen Thread hat ebenfalls eine lineare Komplexität bezogen auf die Anzahl der Tasks. Zur Anzeige eines Speicherzugriffsdiagramms ist diese Effizienz ausreichend, da die Zeitstempel nur für den angezeigten Thread berechnet werden müssen. Für eine komplette Data-Race-Analyse über alle Threads ist das bisherige Verfahren dagegen verbesserungswürdig, da die Laufzeit für Anwendungen mit großer Prozesszahl mehrere Stunden betragen kann. Trotzdem sind im Resultat die Komponenten des entwickelten Prototyps bereits so effizient, dass ein praktikabler Einsatz für viele GASPI-Anwendungen möglich ist.

7 Zusammenfassung und Ausblick

Prediction is very difficult, especially about the future. [Boh18]

Die vorliegende Arbeit erweitert das Portfolio an verfügbaren Analysemethoden für PGAS-Programme. Das geschaffene Werkzeug bietet neue Möglichkeiten, um alle Speicherzugriffe auf gemeinsam genutzte Adressbereiche in ihrem jeweiligen logischen Kontext zu untersuchen und daraus Schlüsse auf die Korrektheit und Effizienz des Programms abzuleiten. Die dazu entwickelten Modelle und Anwendungskonzepte erweisen sich dabei im Vergleich zu bisherigen Arbeiten als vielseitiger einsetzbar und benutzerfreundlicher. Mit dem in Kapitel 4 eingeführten Task-Graph-Modell können PGAS-Operationen präzise abgebildet werden. Dieses Modell bietet die bisher einzigartige Möglichkeit, hybrid parallelisierte Programme unter Einbeziehung aller Synchronisationsoperationen und der lokalen Speicherzugriffe zu analysieren. Das in Kapitel 5 entwickelte Konzept des Speicherzugriffsdiagramms erlaubt erstmalig die Veranschaulichung der Interaktionen von asynchronen und synchronen Speicherzugriffen im gemeinsamen Adressraum eines PGAS-Programms.
In den folgenden drei Abschnitten werden die wissenschaftlich-technischen Beiträge der Arbeit kurz erläutert. Abschließend wird ein Ausblick auf mögliche zukünftige Entwicklungen gegeben, für die diese Arbeit ein Ausgangspunkt sein kann.

7.1 Modellierung von Programmausführungen

In Kapitel 4 wird festgestellt, dass die Berechnung eines Task Graphen aus einer Programmausführung nicht mehr NP-vollständig ist, wenn statt der Synchronisationsoperationen *WAIT* und *CLEAR* die zusammengesetzte Operation *WAITCLEAR* verwendet wird. Das Ergebnis einer solchen Berechnung bildet die logisch notwendigen happened-before-Relationen zwischen den Threads des Programms ab. Außerdem können während dieser Berechnung Synchronisations-Races festgestellt werden. In der Praxis ist das *POST*/*WAITCLEAR*-Synchronisationsmodell nur wenig einschränkend, da eine Vielzahl relevanter Synchronisationsoperationen wie z. B. Kollektive auf dieses Modell zurückgeführt werden können.
Die Entwicklung des *POST*/*WAITCLEAR*-Synchronisationsmodells lieferte auch einen wichtigen Beitrag zur Definition des GASPI-APIs. Der Rückfluss von Erkenntnissen aus der Modellentwicklung in das Design des GASPI-APIs führte zur Einführung von GASPI-Funktionen, die die *WAITCLEAR*-Semantik abbilden. Deswegen ist es auch eine wichtige Feststellung dieser Arbeit, dass das während der Entwicklung von Analysewerkzeugen gewonnene Wissen in die Entwicklung der zu analysierenden Programmiermodelle zurückfließen sollte.
In Kapitel 4 wurde weiterhin gezeigt, dass Task Graphen eine geeignete Grundlage für die Modellierung von Systemen mit asynchronen Operationen bilden. Mit Hilfe von virtuellen Tasks lassen

sich asynchrone Kommunikationsvorgänge so abstrahieren, dass der Task Graph die kausalen Beziehungen zwischen allen Programmereignissen präzise abbilden kann. Da die asynchronen Speicherzugriffe in einem solchen Task Graphen durch die happened-before-Relation mittelbar mit den direkten Speicherzugriffen des Programms verknüpft sind, kann er zur Ermittlung von data races eingesetzt werden.

7.2 Analysekonzepte für PGAS-Programme

Kapitel 5 erarbeitet ein neuartiges Konzept für ein Analysewerkzeug speziell für PGAS-Programme. Der zentrale Bestandteil dieses Konzeptes ist das Speicherzugriffsdiagramm, mit dem das Zusammenspiel asynchroner und synchroner Speicherzugriffe im gemeinsam genutzten PGAS-Adressraum visualisiert werden kann. Speicherzugriffsdiagramme können auf vielfältige Weise zur Untersuchung von Programmeigenschaften eingesetzt werden. Sie veranschaulichen data races, sind aber auch für Effizienzanalysen geeignet.
Durch die interaktive Verknüpfung von Speicherzugriffsdiagrammen mit dem Task-Graph-Modell ist eine detaillierte Untersuchung der logischen Zusammenhänge in einem Programm möglich. Auf diese Weise erschließt diese Arbeit dem Programmierer eine neue Analyseperspektive. Diese mediale Aufbereitung kann auch in Dokumentationen oder in der Lehre zu einem besseren Verständnis der oft komplexen Programm-Zusammenhänge beitragen.

7.3 Effiziente technische Realisierung

Anhand des in Kapitel 6 vorgestellten Werkzeugs konnte gezeigt werden, dass die entwickelten Methoden den anspruchsvollen Anforderungen von HPC-Anwendungen an Laufzeit- und Speicherplatzeffizienz genügen. Ein bedeutender Faktor für die Skalierbarkeit der Task-Graph-Analyse stellt die Einführung der topologischen Sortierung dar, wodurch der Test auf happened-before-Relationen quasi unabhängig von der Anzahl der Prozesse und der Länge der untersuchten Programmläufe wird. Die Aufzeichnung insbesondere der direkten Speicherzugriffe eines Programmlaufs konnte mithilfe des PIN-APIs so effizient gestaltet werden, dass die Verlangsamung der Messung in praktischen Anwendungsfällen vertretbar bleibt.

7.4 Ausblick

Die in dieser Arbeit entwickelten Verfahren können als Ausgangspunkt für verschiedene weiterführende Arbeiten verwendet werden. Das Synchronisationsmodell kann auch auf MPI-Programme angewendet werden. Die *POST*-Operation entspricht `MPI_Send` und die *WAITCLEAR*-Operation `MPI_Recv` mit der Quelle `MPI_ANY_SOURCE`. Um eine Unterscheidung zwischen einem konkreten Quell-Rank und `MPI_ANY_SOURCE` zu erreichen, fasst man die Flags als Mengen auf. Ein Synchronisations-Race bzw. *message race* tritt dann unter den in Abschnitt 4.1.2 (Seite 37) beschriebenen Kriterien auf, wenn nicht-disjunkte Flag-Mengen miteinander in Konflikt stehen. Mit dieser Erweiterung der Theorie kann dann ein Verfahren zum *message matching* von

MPI-Programmen entwickelt werden, das unabhängig von den Zeitstempeln der aufgezeichneten Ereignisse die logischen Synchronisationsbeziehungen zwischen Prozessen ermitteln kann.
Da das entwickelte Modell nicht an den GASPI-Standard geknüpft ist, kann die Behandlung anderer PGAS-APIs wie OpenSHMEM oder MPI-3 sehr einfach integriert werden. Darüber hinaus ist es auch möglich, eine Analyse für heterogene Systeme mit asynchronen Speicherzugriffen (z. B. CUDA) zu realisieren. Das Werkzeug zur Aufzeichnung von Programmereignissen kann durch eine auf die Anforderungen in großen verteilten Systemen angepasste Software, wie z. B. Score-P [KRM+12] ersetzt werden. Hierbei ist auch zu untersuchen, inwieweit eine Quellcode-Instrumentierung der momentan verwendeten binären Instrumentierung vorzuziehen ist. Mit Hilfe einer Quellcode-Instrumentierung kann z. B. eine Datenfluss-Analyse durchgeführt werden, um so Variablenzugriffe, die auf keinen Fall in den gemeinsam genutzten PGAS-Adressraum führen, von vornherein auszuschließen. Demgegenüber steht jedoch die Herausforderung, auch Speicherzugriffe, die in vor-compilierten Bibliotheksfunktionen stattfinden, zu erfassen.
Das im Rahmen der Arbeit implementierte Analysewerkzeug lässt sich auf vielfältige Art erweitern. Eine bereits erfolgte Erweiterung ist die in [HKK17] dokumentierte Ermittlung des kritischen Pfades in einer PGAS-Programmausführung. Die präzise Ermittlung eines solchen Pfades unter Einbeziehung der asynchronen Kommunikationsoperationen ist nur mit den in Kapitel 4 eingeführten virtuellen Tasks möglich.
Das Werkzeug wurde vom Autor unabhängig von dieser Arbeit für verschiedene Analyseaufgaben am Fraunhofer-Institut für Techno- und Wirtschaftsmathematik Kaiserslautern und am Deutschen Zentrum für Luft- und Raumfahrt eingesetzt. Trotzdem handelt es sich hierbei nur um einen Prototyp. Die Resonanz auf die Analyseergebnisse zeigt jedoch, dass in der Zukunft ein Software-Produkt erstrebenswert ist, in das die in dieser Arbeit entwickelten Methoden zur Analyse von PGAS-Programmen einfließen.

Literaturverzeichnis

[ABB+13] Alrutz, Thomas ; Backhaus, Jan ; Brandes, Thomas ; End, Vanessa ; Gerhold, Thomas ; Geiger, Alfred ; Grünewald, Daniel ; Heuveline, Vincent ; Jägersküpper, Jens ; Knüpfer, Andreas ; Krzikalla, Olaf ; Kügeler, Edmund ; Lojewski, Carsten ; Lonsdale, Guy ; Müller-Pfefferkorn, Ralph ; Nagel, Wolfgang ; Oden, Lena ; Pfreundt, Franz-Josef ; Rahn, Mirko ; Sattler, Michael ; Schmidtobreick, Mareike ; Schiller, Annika ; Simmendinger, Christian ; Soddemann, Thomas ; Sutmann, Godehard ; Weber, Henning ; Weiss, Jan-Philipp: GASPI – A Partitioned Global Address Space Programming Interface. In: Keller, Rainer (Hrsg.) ; Kramer, David (Hrsg.) ; Weiss, Jan-Philipp (Hrsg.): *Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–35893–7, 135–136

[AHB03] Artho, Cyrille ; Havelund, Klaus ; Biere, Armin: High-Level Data Races. In: *Journal on Software Testing, Verification & Reliability (STVR)*, 2003

[All18] *Allinea DDT: The debugger for C, C++ and Fortran threaded and parallel code.* `https://www.allinea.com/products/ddt/`, 2018

[AMS12] Abe, T. ; Maeda, T. ; Sato, M.: Model Checking with User-Definable Abstraction for Partitioned Global Address Space Languages. In: *Proceedings of the 6th International Conference on PGAS Programming Models*. Santa Barbara, CA, USA, 2012

[BCA+06] Barton, Christopher ; Caşcaval, Călin ; Almási, George ; Zheng, Yili ; Farreras, Montse ; Chatterje, Siddhartha ; Amaral, José N.: Shared Memory Programming for Large Scale Machines. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2006 (PLDI '06). – ISBN 1–59593–320–4, 108–117

[BCO05] Berdine, Josh ; Calcagno, Cristiano ; O'Hearn, Peter W.: Symbolic Execution with Separation Logic. In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. Berlin, Heidelberg : Springer-Verlag, 2005 (APLAS'05). – ISBN 3–540–29735–9, 978–3–540–29735–2, 52–68

[BCOP05] Bornat, Richard ; Calcagno, Cristiano ; O'Hearn, Peter ; Parkinson, Matthew: Permission Accounting in Separation Logic. In: *SIGPLAN Not.* 40 (2005), Januar, Nr. 1, 259–270. `http://dx.doi.org/10.1145/1047659.1040327`. – DOI 10.1145/1047659.1040327. – ISSN 0362–1340

[BDDP11] Botincan, Matko ; Dodds, Mike ; Donaldson, Alastair F. ; Parkinson, Matthew J.: Safe asynchronous multicore memory operations. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2011 (ASE '11). – ISBN 978–1–4577–1638–6, S. 153–162

[BE06] Brodersen, Olaf ; Eisfeld, Bernhard: DPW-3 DLR F6/F6-FX2B Results. In: *3rd AIAA Drag Prediction Workshop*, AIAA, Juni 2006

[Ber66] Bernstein, A. J.: Analysis of Programs for Parallel Processing. In: *IEEE Transactions on Electronic Computers* EC-15 (1966), Oct, Nr. 5, S. 757–763. `http://dx.doi.org/10.1109/PGEC.1966.264565`. – DOI 10.1109/PGEC.1966.264565. – ISSN 0367–7508

[Ber08] Berg, M. de: *Computational Geometry: Algorithms and Applications*. Springer, 2008 `https://books.google.de/books?id=tkyG8W2163YC`. – ISBN 9783540779735

[BH15] Belli, R. ; Hoefler, T.: Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In: *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*, IEEE, May 2015

[BK94] Barriuso, Ray ; Knies, Allan: *SHMEM user's guide for C*. June 1994

[BKS+11] Basermann, Achim ; Kersken, Hans-Peter ; Schreiber, Andreas ; Gerhold, Thomas ; Jägersküpper, Jens ; Kroll, Norbert ; Backhaus, Jan ; Kügeler, Edmund ; Alrutz, Thomas ; Simmendinger, Christian ; Krzikalla, Olaf u. a.: HICFD: highly efficient implementation of CFD codes for HPC Many-Core architectures. In: *Competence in High Performance Computing 2010*. Springer, Berlin, Heidelberg, 2011, S. 1–13

[Boh18] Bohr, Niels: *Wikiquote: Niels Bohr*. `https://en.wikiquote.org/wiki/Niels_Bohr`, 2018

[BSGT04] Byna, S. ; Sun, Xian-He ; Gropp, W. ; Thakur, R.: Predicting memory-access cost based on data-access patterns. In: *Cluster Computing, 2004 IEEE International Conference on*, 2004. – ISSN 1552–5244, S. 327–336

[CCP+10] Chapman, Barbara ; Curtis, Tony ; Pophale, Swaroop ; Poole, Stephen ; Kuehn, Jeff ; Koelbel, Chuck ; Smith, Lauren: Introducing OpenSHMEM: SHMEM for the PGAS community. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. New York, NY, USA : ACM, 2010 (PGAS '10). – ISBN 978–1–4503–0461–0, S. 2:1–2:3

[CCZ04] Callahan, D. ; Chamberlain, B.L. ; Zima, H.P.: The cascade high productivity language. In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004.*, 2004, S. 52–60

[CDC+99] Carlson, William W. ; Draper, Jesse M. ; Culler, David E. ; Yelick, Kathy ; III, Eugene D. B. ; Warren, Karen: Introduction to UPC and Language Specification. (1999)

[CDMM13] Calin, Georgel ; Derevenetc, Egor ; Majumdar, Rupak ; Meyer, Roland: A Theory of Partitioned Global Address Spaces. In: Seth, Anil (Hrsg.) ; Vishnoi, Nisheeth K. (Hrsg.): *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013)* Bd. 24. Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013 (Leibniz International Proceedings in Informatics (LIPIcs)). – ISBN 978–3–939897–64–4, 127–139

[CDT+14] Chen, Z. ; Dinan, J. ; Tang, Z. ; Balaji, P. ; Zhong, H. ; Wei, J. ; Huang, T. ; Qin, F.: MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications. In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014. – ISSN 2167–4329, S. 499–510

[CGS+05] Charles, Philippe ; Grothoff, Christian ; Saraswat, Vijay ; Donawa, Christopher ; Kielstra, Allan ; Ebcioglu, Kemal ; Praun, Christoph von ; Sarkar, Vivek: X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 2005 (OOPSLA '05). – ISBN 1–59593–031–0, 519–538

[CL85] Chandy, K M. ; Lamport, Leslie: Distributed snapshots: determining global states of distributed systems. In: *ACM Transactions on Computer Systems (TOCS)* 3 (1985), Nr. 1, S. 63–75

[Cla06] Claus, Volker: *Duden Informatik A - Z : Fachlexikon für Studium, Ausbildung und Beruf /*. 4. Aufl. Mannheim ; , Leipzig ; , Wien ; , Zürich : : Dudenverl.„ 2006 `http://slubdd.de/katalog?TN_libero_mab214262748`. – ISBN 3411052341

[CLC+12] Chen, Z. ; Li, X. ; Chen, J. Y. ; Zhong, H. ; Qin, F.: SyncChecker: Detecting Synchronization Errors between MPI Applications and Libraries. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012. – ISSN 1530–2075, S. 342–353

[Col11] Collet, Y.: *LZ4*. `http://lz4.github.io/lz4/`, 2011

[Coo03] Cooling, J.E.: *Software Engineering for Real-time Systems*. Addison-Wesley, 2003 `https://books.google.de/books?id=xqPJjzDsK-AC`. – ISBN 9780201596205

[Cou04] Council, North American Electric R.: *Technical Analysis of the August 14, 2003 Blackout: What Happened, Why, and what Did We Learn?* North American Electric Reliability Council, 2004 (Engineering case studies online). `https://books.google.de/books?id=ZfJ7MwEACAAJ`

[DAC+14] Dinh, Minh N. ; Abramson, David ; Chao, Jin ; DeRose, Luiz ; Moench, Bob ; Gontarek, Andrew: Supporting Relative Debugging for Large-scale UPC Programs. In: *Procedia Computer Science* 29 (2014), 1491 - 1503. http://dx.doi.org/https://doi.org/10.1016/j.procs.2014.05.135. – DOI https://doi.org/10.1016/j.procs.2014.05.135. – ISSN 1877–0509. – 2014 International Conference on Computational Science

[DAS12] Dubois, M. ; Annavaram, M. ; Stenström, P.: *Parallel Computer Organization and Design*. Cambridge University Press, 2012 (Parallel Computer Organization and Design). – ISBN 9780521886758

[Den05] Denning, Peter J.: The Locality Principle. In: *Commun. ACM* 48 (2005), Juli, Nr. 7, 19–24. http://dx.doi.org/10.1145/1070838.1070856. – DOI 10.1145/1070838.1070856. – ISSN 0001–0782

[Der13] Derevenetc, Egor: Private Communication. (2013)

[Dij65] Dijkstra, E. W.: Solution of a Problem in Concurrent Programming Control. In: *Commun. ACM* 8 (1965), September, Nr. 9, 569. http://dx.doi.org/10.1145/365559.365617. – DOI 10.1145/365559.365617. – ISSN 0001–0782

[Dij72] Dijkstra, Edsger W.: Structured Programming. Version: 1972. http://dl.acm.org/citation.cfm?id=1243380.1243381. London, UK, UK : Academic Press Ltd., 1972. – ISBN 0–12–200550–3, Kapitel Chapter I: Notes on Structured Programming, 1–82

[DLHV16] Dan, Andrei M. ; Lam, Patrick ; Hoefler, Torsten ; Vechev, Martin: Modeling and Analysis of Remote Memory Access Programming. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 2016 (OOPSLA 2016). – ISBN 978–1–4503–4444–9, 129–144

[DP12] Denninger, Oliver ; Padberg, Frank: *Typische Fehler in parallelen Programmen*. Mai 2012. – Parallel 2012. Softwarekonferenz und Workshops für Parallel Programming, Concurrency und Multicore-Systeme.

[DRW06] Duboc, Leticia ; Rosenblum, David S. ; Wicks, Tony: A Framework for Modelling and Analysis of Software Systems Scalability. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA : ACM, 2006 (ICSE '06). – ISBN 1–59593–375–1, 949–952

[DS91] Denert, E. ; Siedersleben, J.: *Software-Engineering: methodische Projektabwicklung*. Springer-Verlag, 1991 https://books.google.de/books?id=LfORAQAAMAAJ. – ISBN 9783540534044

[Ebn11] Ebnenasir, Ali: UPC-SPIN: A Framework for the Model Checking of UPC Programs. In: *Proceedings of the 5th International Conference on PGAS Programming Models*. Galveston Island, TX, USA, 2011

[ECD10] Einstein, A. ; Calaprice, A. ; Dyson, F.: *The Ultimate Quotable Einstein*. Princeton University Press, 2010 https://books.google.de/books?id=G_iziBAPXtEC. – ISBN 9781400835966

[EGCSY05] El-Ghazawi, T. ; Carlson, W. ; Sterling, T. ; Yelick, K.: *UPC: Distributed Shared Memory Programming*. Wiley, 2005 (Wiley Series on Parallel and Distributed Computing). https://books.google.de/books?id=n4pknjxmh7EC. – ISBN 9780471478379

[EGP89] Emrath, Perry A. ; Ghosh, Sanjoy ; Padua, David A.: Event Synchronization Analysis for Debugging Parallel Programs. In: *Proceedings of Supercomputing '89*, 1989, S. 580–588

[EGP92] Emrath, P.A. ; Ghosh, S. ; Padua, D.A.: Detecting nondeterminacy in parallel programs. In: *Software, IEEE* 9 (1992), January, Nr. 1, S. 69 –77. http://dx.doi.org/10.1109/52.108783. – DOI 10.1109/52.108783. – ISSN 0740–7459

[Exa16] *The Challenges of Exascale*. Website, 2016. – Available online at http://science.energy.gov/ascr/research/scidac/exascale-challenges/; visited on July 2016.

[FF09] Flanagan, Cormac ; Freund, Stephen N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2009 (PLDI '09). – ISBN 978–1–60558–392–1, 121–133

[Fon12] Fontane, T.: *Effi Briest*. outlook Verlag, 2012 https://books.google.de/books?id=uAv4w2zSRfoC. – ISBN 9783864038457

[For12] Forum, Message Passing I.: *MPI: A Message-Passing Interface Standard Version 3.0*. Sep. 2012. – Chapter author for Collective Communication, Process Topologies, and One Sided Communications

[GAM95] Gupta, A. ; Anderson, T. E. ; Martonosi, M.: Tuning Memory Performance of Sequential and Parallel Programs. In: *Computer* 28 (1995), 04, 32-40. http://dx.doi.org/10.1109/2.375175. – DOI 10.1109/2.375175. – ISSN 0018–9162

[GAS11] *GASPI – Global Adress Space Programming Interface*. Website, 2011. – Available online at http://www.gaspi.de; visited on January 29th 2015.

[Gel98] Gelernter, D.H.: *Machine Beauty: Elegance and the Heart of Technology*. Basic Books, 1998 (Master Minds Series). https://books.google.de/books?id=ntZQAAAAMAAJ. – ISBN 9780465045167

[GGJ+17] Gimenez, A. A. ; Gamblin, T. ; Jusufi, I. ; Bhatele, A. ; Schulz, M. ; Bremer, P. T. ; Hamann, B.: MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors. In: *IEEE Transactions on Visualization*

and Computer Graphics PP (2017), Nr. 99, S. 1–1. http://dx.doi.org/10.1109/TVCG.2017.2718532. – DOI 10.1109/TVCG.2017.2718532. – ISSN 1077–2626

[GPI12] *The Fraunhofer GPI programming model.* Website, 2012. – Available online at http://www.gaspi.de/uploads/media/The_Fraunhofer_GPI_Programming_Model_Gruenewald_ITWM_01.pdf; visited on January 29th 2014.

[Gre06] Greenberg, H.J.: *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at INFORMS 2004, Denver, CO.* Springer New York, 2006 (International Series in Operations Research & Management Science). https://books.google.de/books?id=aWG6hCx2hMQC. – ISBN 9780387228273

[GV95] Grabner, Siegfried ; Volkert, Jens: Debugging parallel programs using event graph manipulation. In: *Computing Systems in Engineering* 6 (1995), Nr. 4, 443 - 450. http://dx.doi.org/http://dx.doi.org/10.1016/0956-0521(95)00040-2. – DOI http://dx.doi.org/10.1016/0956–0521(95)00040–2. – ISSN 0956–0521

[GZH+94] Gerstel, O. ; Zaks, S. ; Hurfin, M. ; Plouzeau, N. ; Raynal, M.: On-the-fly replay: a practical paradigm and its implementation for distributed debugging. In: *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*, 1994, S. 266–272

[Her15] Herold, Christian: *Optimierung des Kommunikationsverhaltens von parallelen Anwendungen mit einseitigen Kommunikationsprimitiven.* Dresden, Hochschule für Technik und Wirtschaft, Master-Arbeit, 2015

[Hil90] Hill, Mark D.: What is Scalability? In: *SIGARCH Comput. Archit. News* 18 (1990), Dezember, Nr. 4, 18–21. http://dx.doi.org/10.1145/121973.121975. – DOI 10.1145/121973.121975. – ISSN 0163–5964

[HKK17] Herold, C. ; Krzikalla, O. ; Knüpfer, A.: Optimizing One-Sided Communication of Parallel Applications Using Critical Path Methods. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, S. 567–576

[HPS+12] Hilbrich, Tobias ; Protze, Joachim ; Schulz, Martin ; Supinski, Bronis R. ; Müller, Matthias S.: MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* Los Alamitos, CA, USA : IEEE Computer Society Press, 2012 (SC '12). – ISBN 978–1–4673–0804–5, 30:1–30:11

[HSN+13] Hilbrich, T. ; Supinski, B. R. ; Nagel, W. E. ; Protze, J. ; Baier, C. ; Müller, M. S.: Distributed wait state tracking for runtime MPI deadlock detection. In: *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. – ISSN 2167–4329, S. 1–12

[IBM17] *IBM Terminology*. Website, 2017. – Available online at `http://www-01.ibm.com/software/globalization/terminology/t.html`; visited on May 2017.

[iee90] IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), Dec, S. 1–84. `http://dx.doi.org/10.1109/IEEESTD.1990.101064`. – DOI 10.1109/IEEESTD.1990.101064

[ISO12] ISO: *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland : International Organization for Standardization, 2012. – 1338 (est.) S. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`

[JNPS09] Joshi, Pallavi ; Naik, Mayur ; Park, Chang-Seo ; Sen, Koushik: CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In: *Proceedings of the 21st International Conference on Computer Aided Verification*. Berlin, Heidelberg : Springer-Verlag, 2009 (CAV '09). – ISBN 978–3–642–02657–7, 675–681

[JSC14] Jana, Siddhartha ; Schuchart, Joseph ; Chapman, Barbara: Analysis of Energy and Performance of PGAS-based Data Access Patterns. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. New York, NY, USA : ACM, 2014 (PGAS '14). – ISBN 978–1–4503–3247–7, 15:1–15:10

[Kah65] Kahan, W.: Pracniques: Further Remarks on Reducing Truncation Errors. In: *Commun. ACM* 8 (1965), Januar, Nr. 1, 40–. `http://dx.doi.org/10.1145/363707.363723`. – DOI 10.1145/363707.363723. – ISSN 0001–0782

[KBD+08] Knüpfer, Andreas ; Brunst, Holger ; Doleschal, Jens ; Jurenz, Matthias ; Lieber, Matthias ; Mickler, Holger ; Müller, Matthias S. ; Nagel, Wolfgang E.: The Vampir Performance Analysis Tool-Set. In: Resch, Michael (Hrsg.) ; Keller, Rainer (Hrsg.) ; Himmler, Valentin (Hrsg.) ; Krammer, Bettina (Hrsg.) ; Schulz, Alexander (Hrsg.): *Tools for High Performance Computing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–68564–7, S. 139–155

[KKMPN13] Krzikalla, Olaf ; Knüpfer, Andreas ; Müller-Pfefferkorn, Ralph ; Nagel, Wolfgang: On the Modelling of One-Sided Communication Systems. In: *Proceedings of the 7th International Conference on PGAS Programming Models*. Edinburgh, UK, October 2013, S. 41–53

[KMPN16] Krzikalla, Olaf ; Müller-Pfefferkorn, Ralph ; Nagel, Wolfgang E.: Synchronization Debugging of Hybrid Parallel Programs. In: Dutot, Pierre-François (Hrsg.) ; Trystram, Denis (Hrsg.): *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. Cham : Springer International Publishing, 2016. – ISBN 978–3–319–43659–3, 37–50

[Kra00] Kranzlmüller, Dieter: *Event Graph Analysis For Debugging Massively Parallel Programs*. Joh. Kepler University Linz, Austria, Diss., 2000

[KRM+12] Knüpfer, Andreas ; Rössel, Christian ; Mey, Dieter a. ; Biersdorff, Scott ; Diethelm, Kai ; Eschweiler, Dominic ; Geimer, Markus ; Gerndt, Michael ; Lorenz, Daniel ; Malony, Allen ; Nagel, Wolfgang E. ; Oleynik, Yury ; Philippen, Peter ; Saviankou, Pavel ; Schmidl, Dirk ; Shende, Sameer ; Tschüter, Ronny ; Wagner, Michael ; Wesarg, Bert ; Wolf, Felix: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In: Brunst, Holger (Hrsg.) ; Müller, Matthias S. (Hrsg.) ; Nagel, Wolfgang E. (Hrsg.) ; Resch, Michael M. (Hrsg.): *Tools for High Performance Computing 2011*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–31476–6, S. 79–91

[Krz15] Krzikalla, Olaf: *GASPI proposal: Memory Ordering*. Website, 2015. – Available online at `http://www.gaspi.de/proposals/memory_model.pdf`; visited on April 2017.

[Lam78] Lamport, Leslie: Time, clocks, and the ordering of events in a distributed system. In: *Commun. ACM* 21 (1978), Juli, Nr. 7, S. 558–565. `http://dx.doi.org/10.1145/359545.359563`. – DOI 10.1145/359545.359563. – ISSN 0001–0782

[LCM+05] Luk, Chi-Keung ; Cohn, Robert ; Muth, Robert ; Patil, Harish ; Klauser, Artur ; Lowney, Geoff ; Wallace, Steven ; Reddi, Vijay J. ; Hazelwood, Kim: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2005 (PLDI '05). – ISBN 1–59593–056–6, 190–200

[LMC87] Leblanc, T. J. ; Mellor-Crummey, J. M.: Debugging Parallel Programs with Instant Replay. In: *IEEE Transactions on Computers* C-36 (1987), April, Nr. 4, S. 471–482. `http://dx.doi.org/10.1109/TC.1987.1676929`. – DOI 10.1109/TC.1987.1676929. – ISSN 0018–9340

[LMC13] Liu, Xu ; Mellor-Crummey, John: A Data-centric Profiler for Parallel Programs. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : ACM, 2013 (SC '13). – ISBN 978–1–4503–2378–9, 28:1–28:12

[LSZ90] Leu, E. ; Schiper, A. ; Zramdini, A.: Execution replay on distributed memory architectures. In: *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990*, 1990, S. 106–112

[LT93] Leveson, N. G. ; Turner, C. S.: An Investigation of the Therac-25 Accidents. In: *Computer* 26 (1993), Juli, Nr. 7, 18–41. `http://dx.doi.org/10.1109/MC.1993.274940`. – DOI 10.1109/MC.1993.274940. – ISSN 0018–9162

[Mat88] Mattern, Friedemann: Virtual Time and Global States of Distributed Systems. In: *PARALLEL AND DISTRIBUTED ALGORITHMS*, North-Holland, 1988, S. 215–226

[MDD+89] Miner, J.G. ; Dean, D. ; Decuir, J.C. ; Nicholson, R.H. ; Tanaka, A.: *Personal computer apparatus for block transfer of bit-mapped image data.* `https://www.google.com/patents/US4874164`. Version: Oktober 17 1989. – US Patent 4,874,164

[MRH14] Milthorpe, Josh ; Rendell, Alistair P. ; Huber, Thomas: PGAS-FMM: Implementing a Distributed Fast Multipole Method Using the X10 Programming Language. In: *Concurr. Comput. : Pract. Exper.* 26 (2014), März, Nr. 3, 712–727. `http://dx.doi.org/10.1002/cpe.3039`. – DOI 10.1002/cpe.3039. – ISSN 1532–0626

[NBDK96] Netzer, Robert H. ; Brennan, Timothy W. ; Damodaran-Kamal, Suresh K.: Debugging race conditions in message-passing programs. In: *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* ACM, 1996, S. 31–40

[Net04] Nethercote, Nicholas: *Dynamic Binary Analysis and Instrumentation.* University of Cambridge, United Kingdom, Diss., 2004

[NM90] Netzer, Robert H. ; Miller, Barton P.: On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. In: *Proceedings of the 1990 International Conference on Parallel Processing*, 1990, S. 93–97

[NM92a] Netzer, R. H. B. ; Miller, B. P.: Optimal Tracing and Replay for Debugging Message-passing Parallel Programs. In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1992 (Supercomputing '92). – ISBN 0–8186–2630–5, 502–511

[NM92b] Netzer, Robert H. B. ; Miller, Barton P.: What Are Race Conditions?: Some Issues and Formalizations. In: *ACM Lett. Program. Lang. Syst.* 1 (1992), März, Nr. 1, 74–88. `http://dx.doi.org/10.1145/130616.130623`. – DOI 10.1145/130616.130623. – ISSN 1057–4514

[NR98] Numrich, Robert W. ; Reid, John: Co-array Fortran for Parallel Programming. In: *SIGPLAN Fortran Forum* 17 (1998), August, Nr. 2, 1–31. `http://dx.doi.org/10.1145/289918.289920`. – DOI 10.1145/289918.289920. – ISSN 1061–7264

[NS07] Nethercote, Nicholas ; Seward, Julian: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA : ACM, 2007 (PLDI '07). – ISBN 978–1–59593–633–2, 89–100

[NWT+07] Narayanasamy, Satish ; Wang, Zhenghao ; Tigani, Jordan ; Edwards, Andrew ; Calder, Brad: Automatically Classifying Benign and Harmful Data Races

Using Replay Analysis. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2007 (PLDI '07). – ISBN 978–1–59593–633–2, 22–31

[OA10] Ortega-Arjona, J.L.: *Patterns for Parallel Software Design*. Wiley, 2010 (Wiley Software Patterns Series). https://books.google.de/books?id=u6Bgh1nfZvUC. – ISBN 9780470970874

[O'H07] O'Hearn, Peter W.: Resources, concurrency, and local reasoning. In: *Theoretical Computer Science* 375 (2007), Nr. 1, 271 - 307. http://dx.doi.org/http://dx.doi.org/10.1016/j.tcs.2006.12.035. – DOI http://dx.doi.org/10.1016/j.tcs.2006.12.035. – ISSN 0304–3975

[OMGB10] Ostadzadeh, S. A. ; Meeuws, Roel J. ; Galuzzi, Carlo ; Bertels, Koen: QUAD – A Memory Access Pattern Analyser. In: Sirisuk, Phaophak (Hrsg.) ; Morgan, Fearghal (Hrsg.) ; El-Ghazawi, Tarek (Hrsg.) ; Amano, Hideharu (Hrsg.): *Reconfigurable Computing: Architectures, Tools and Applications: 6th International Symposium, ARC 2010, Bangkok, Thailand, March 17-19, 2010. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–12133–3, 269–281

[OPE13] *OpenSHMEM*. Website, 2013. – Available online at http://www.openshmem.org; visited on December 20th 2013.

[OS98] Oppenheim, A.V. ; Schafer, R.W.: *Zeitdiskrete Signalverarbeitung*. De Gruyter, 1998 (Grundlagen der Schaltungstechnik). https://books.google.de/books?id=u6cDDgAAQBAJ. – ISBN 9783486792966

[OTF15] *OTF2 Knowledge: How to Match MPI Messages*. http://blog.automaton2000.com/2015/07/, 2015

[Par12] Park, Chang S.: *Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems*, EECS Department, University of California, Berkeley, Diss., December 2012

[Pat11] Patterson, D.A.: *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2011 (The Morgan Kaufmann Series in Computer Architecture and Design). https://books.google.de/books?id=gQ-fSqbLfFoC. – ISBN 9780123838735

[PGA13] *Partitioned Global Address Space*. Website, 2013. – Available online at http://www.pgas.org; visited on December 20th 2013.

[PKS02] Pol, M. ; Koomen, T. ; Spillner, A.: *Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap*. dpunkt-Verlag, 2002 https://books.google.de/books?id=RDy4ygAACAAJ. – ISBN 9783898641562

[PS07] Pozniansky, Eli ; Schuster, Assaf: MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. In: *Concurr. Comput. : Pract. Exper.* 19 (2007), März, Nr. 3, 327–340. http://dx.doi.org/10.1002/cpe.v19:3. – DOI 10.1002/cpe.v19:3. – ISSN 1532–0626

[PSHI11] Park, Chang-Seo ; Sen, Koushik ; Hargrove, Paul ; Iancu, Costin: Efficient data race detection for distributed memory parallel programs. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : ACM, 2011 (SC '11). – ISBN 978–1–4503–0771–0, S. 51:1–51:12

[qt118] *Qt - Software development made smarter*. https://www.qt.io/, 2018

[Rat16] Rathmann, U.: *Qwt - Qt Widgets for Technical Applications*. http://qwt.sourceforge.net/, 2016

[Ray12] Raynal, M.: *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Berlin Heidelberg, 2012 https://books.google.de/books?id=0a1AAAAAQBAJ. – ISBN 9783642320279

[RB11] Rane, A. ; Browne, J.: Performance Optimization of Data Structures Using Memory Access Characterization. In: *2011 IEEE International Conference on Cluster Computing*, 2011. – ISSN 1552–5244, S. 570–574

[RDC00] Ronsse, M. ; De Bosschere, K. ; Chassin de Kergommeaux, J.: Execution replay and debugging. In: *eprint arXiv:cs/0011006* (2000), November

[Rev16] Reviews, CTI: *Mathematical Structures for Computer Science, A Modern Approach to Discrete Mathematics*. Cram101, 2016 https://books.google.de/books?id=VUYfJIZzUTEC. – ISBN 9781619069220

[RHJ09] Rabenseifner, R. ; Hager, G. ; Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009. – ISSN 1066–6192, S. 427–436

[RM95] Ramanujam, J ; Mathew, A: Analysis of event synchronization in parallel programs. In: *Languages and Compilers for Parallel Computing*. Springer, 1995, S. 300–315

[RZR+15] Radulovic, Milan ; Zivanovic, Darko ; Ruiz, Daniel ; Supinski, Bronis R. ; McKee, Sally A. ; Radojković, Petar ; Ayguadé, Eduard: Another Trip to the Wall: How Much Will Stacked DRAM Benefit HPC? In: *Proceedings of the 2015 International Symposium on Memory Systems*. New York, NY, USA : ACM, 2015 (MEMSYS '15). – ISBN 978–1–4503–3604–8, 31–36

[RZS+12] Raman, Raghavan ; Zhao, Jisheng ; Sarkar, Vivek ; Vechev, Martin ; Yahav, Eran: Scalable and precise dynamic datarace detection for structured parallelism. In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2012 (PLDI '12). – ISBN 978–1–4503–1205–9, S. 531–542

[SABW13] Seufert, S. ; Anand, A. ; Bedathur, S. ; Weikum, G.: FERRARI: Flexible and efficient reachability range assignment for graph indexing. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013. – ISSN 1063–6382, S. 1009–1020

[SBG08] Su, H. H. ; Billingsley, M. ; George, A. D.: Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008. – ISSN 1530–2075, S. 1–8

[SBN+97] Savage, Stefan ; Burrows, Michael ; Nelson, Greg ; Sobalvarro, Patrick ; Anderson, Thomas: Eraser: a dynamic data race detector for multithreaded programs. In: *ACM Trans. Comput. Syst.* 15 (1997), November, Nr. 4, S. 391–411. `http://dx.doi.org/10.1145/265924.265927`. – DOI 10.1145/265924.265927. – ISSN 0734–2071

[sci17] *SciDAVis - Scientific Data Analysis and Visualization*. `http://scidavis.sourceforge.net/`, 2017

[Sen08] Sen, Koushik: Race Directed Random Testing of Concurrent Programs. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2008 (PLDI '08). – ISBN 978–1–59593–860–2, 11–21

[SFS+11] Subotic, Vladimir ; Ferrer, Roger ; Sancho, Jose C. ; Labarta, Jesús ; Valero, Mateo: Quantifying the Potential Task-based Dataflow Parallelism in MPI Applications. In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*. Berlin, Heidelberg : Springer-Verlag, 2011 (Euro-Par'11). – ISBN 978–3–642–23399–9, 39–51

[SGH06] Schwamborn, Dieter ; Gerhold, Thomas ; Heinrich, Ralf: The DLR TAU-code: Recent applications in research and industry. In: *ECCOMAS CFD 2006 Conference*, 2006. – auf CD

[SH88] Snelling, David F. ; Hoffmann, Geerd-R.: A comparative study of libraries for parallel processing. In: *Parallel Computing* 8 (1988), Nr. 1-3, 255 - 266. `http://dx.doi.org/http://dx.doi.org/10.1016/0167-8191(88)90129-9`. – DOI http://dx.doi.org/10.1016/0167–8191(88)90129–9. – ISSN 0167–8191. – Proceedings of the International Conference on Vector and Parallel Processors in Computational Science {III}

[SH11] Schöne, Robert ; Hackenberg, Daniel: On-line Analysis of Hardware Performance Events for Workload Characterization and Processor Frequency Scaling Decisions. In: *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA : ACM, 2011 (ICPE '11). – ISBN 978–1–4503–0519–8, 481–486

[Sim14a] Simmendinger, C.: *CFD Proxy Version 1.0.1*. `https://github.com/PGAS-community-benchmarks/CFD-Proxy`, 2014

[Sim14b] Simmendinger, C.: *PGAS-community-benchmarks*. `https://github.com/PGAS-community-benchmarks`, 2014

[Sim15] Simmendinger, C.: *Hybrid, asynchronous 3d version of the Synch_p2p benchmark from Intel PRK*. `https://github.com/PGAS-community-benchmarks/Asynch-3D`, 2015

[Sin07] Sinnen, Oliver: *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007. – ISBN 0471735760

[SLG+15] Servat, Harald ; Llort, Germán ; González, Juan ; Giménez, Judit ; Labarta, Jesús: Low-Overhead Detection of Memory Access Patterns and Their Time Evolution. In: Träff, Jesper L. (Hrsg.) ; Hunold, Sascha (Hrsg.) ; Versaci, Francesco (Hrsg.): *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. – ISBN 978–3–662–48096–0, 57–69

[SM11] Schaik, Sebastiaan J. ; Moor, Oege de: A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2011 (SIGMOD '11). – ISBN 978–1–4503–0661–4, 913–924

[SN05] Seward, Julian ; Nethercote, Nicholas: Using Valgrind to Detect Undefined Value Errors with Bit-precision. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005 (ATEC '05), 2–2

[SSO+95] Stricker, T. ; Stichnoth, J. ; O'Hallaron, D. ; Hinrichs, S. ; Gross, T.: Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In: *Proceedings of the 9th International Conference on Supercomputing*. New York, NY, USA : ACM, 1995 (ICS '95). – ISBN 0–89791–728–6, 1–10

[SWS+12] Shan, Hongzhang ; Wright, Nicholas J. ; Shalf, John ; Yelick, Katherine ; Wagner, Marcus ; Wichmann, Nathan: A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and

Comparison with MPI. In: *SIGMETRICS Perform. Eval. Rev.* 40 (2012), Oktober, Nr. 2, 92–98. http://dx.doi.org/10.1145/2381056.2381077. – DOI 10.1145/2381056.2381077. – ISSN 0163–5999

[TK12] Tallent, N. R. ; Kerbyson, D.: Data-centric performance analysis of PGAS applications. In: *Proc. of the Second Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, 2012

[tot18] *Totalview® for HPC User Guide: Viewing Shared Objects.* http://docs.roguewave.com/totalview/2017.0/html/User_Guides/ViewingSharedObjects.html, 2018

[VCJZ14] Veloso, Renê R. ; Cerf, Loïc ; Junior, Wagner M. ; Zaki, Mohammed J.: *Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach*

[VGK+11] Vo, A. ; Gopalakrishnan, G. ; Kirby, R. M. ; Supinski, B. R. ; Schulz, M. ; Bronevetsky, G.: Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011. – ISSN 1089–795X, S. 330–339

[Wer16] Werner, Michael: *Aufzeichnung und Auswertung von nicht-lokalen Speicherzugriffen in NUMA Systemen.* Dresden, Technische Universität, Diplomarbeit, 2016

[Wil90] Wilde, O.: *The Importance of Being Earnest.* Dover Publications, 1990 (Dover thrift editions). https://books.google.de/books?id=sWu2bfKcyb0C. – ISBN 9780486264783

[WKm13] Williams, Thomas ; Kelley, Colin ; many others: *Gnuplot 4.6: an interactive plotting program.* http://gnuplot.sourceforge.net/, April 2013

[WM95] Wulf, Wm. A. ; McKee, Sally A.: Hitting the Memory Wall: Implications of the Obvious. In: *SIGARCH Comput. Archit. News* 23 (1995), März, Nr. 1, 20–24. http://dx.doi.org/10.1145/216585.216588. – DOI 10.1145/216585.216588. – ISSN 0163–5964

[WM14] Wijngaart, R. F. V. ; Mattson, T. G.: The Parallel Research Kernels. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, S. 1–6

[Wol89] Wolfe, Michael: Iteration Space Tiling for Memory Hierarchies. In: *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing.* Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1989. – ISBN 0–89871–228–9, 357–361

[YBD01] Yu, Y. ; Beyls, K. ; D'Hollander, E. H.: Visualizing the impact of the cache on program execution. In: *Proceedings Fifth International Conference on Information Visualisation*, 2001, S. 336–341

[YCZ10] Yildirim, Hilmi ; Chaoji, Vineet ; Zaki, Mohammed J.: GRAIL: Scalable Reachability Index for Large Graphs. In: *Proc. VLDB Endow.* 3 (2010), September, Nr. 1-2, 276–284. http://dx.doi.org/10.14778/1920841.1920879. – DOI 10.14778/1920841.1920879. – ISSN 2150–8097

[YM88] Yang, C. Q. ; Miller, B. P.: Critical path analysis for the execution of parallel and distributed programs. In: *[1988] Proceedings. The 8th International Conference on Distributed*, 1988, S. 366–373

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

CAF Coarray Fortran

CFD Computational Fluid Dynamics

DAG Directed Acyclic Graph

DLR Deutsches Zentrum für Luft- und Raumfahrt

GASPI Global Address Space Programming Interface

HPC High Performance Computing

ITWM Fraunhofer-Institut für Techno- und Wirtschaftsmathematik

MPI Message Passing Interface

NUMA Non-uniform memory access

NVRAM Non-volatile RAM

PGAS Partitioned Global Address Space

RDMA Remote Direct Memory Access

SIMD Single Instruction Multiple Data

SPMD Single Program Multiple Data

UPC Unified Parallel C

ZIH Zentrum für Informationsdienste und Hochleistungsrechnen

Danksagung

Mein besonderer Dank gilt Prof. Dr. Wolfgang E. Nagel, der mir die Möglichkeit zur Erstellung dieser Arbeit gegeben hat. Das ZIH habe ich als ein Institut erlebt, in dem trotz aller wissenschaftspolitischen Zwänge freie Forschung möglich ist.
Des Weiteren möchte ich mich bei Dr. Ralph Müller-Pfefferkorn für die vielen Gespräche bedanken, die wesentlich zum Fortgang dieser Arbeit beigetragen haben. Daneben gebührt Dank Christian Herold für seine begleitenden Arbeiten, Dr. Michael Kluge und Dr. Daniel Molka für ihre hilfreichen Anmerkungen sowie Dr. Richard Grunzke für oft lange Diskussionen. Weiterhin möchte ich mich bei allen Kollegen am ZIH bedanken, die in vielerlei Hinsicht zum Gelingen der Arbeit beigetragen haben.
Vielen Dank an das DLR-Institut SP Dresden und hier vor allem Prof. Dr. Norbert Kroll, der mich bei der Fertigstellung der Arbeit unterstützte. Mein Dank gilt auch den Kollegen des ITWM Kaiserslautern, insbesondere Dr. Mirko Rahn, die mir im Rahmen dieser Arbeit die Möglichkeit gegeben haben, meine Methodik an praktischen Beispielen zu testen und weiterzuentwickeln.
Der abschließende und wichtigste Dank gebührt meiner Lebensgefährtin Ines Schällig, die mich über all die Jahre mal geduldig, mal fordernd motiviert hat und die mir auch nach der Geburt unserer Tochter den Rücken freigehalten hat.